MASTER EN SCIENCES INFORMATIQUES

Validation du protocole TCP Xeno

Kajeguhakwa, Iddi

Award date:
2009

Awarding institution:
Universite de Namur

Ma toute grande reconnaissance va à mon promoteur et directeur de mémoire, le Professeur Laurent Schumacher d'avoir accepté de me superviser et me soutenir pendant ce travail. Ses connaissances, ses conseils, et son suivi attentif ont beaucoup contribués à la réalisation de ce travail.

Mes sincères remerciements vont également à tous ceux qui ont contribué de près ou de loin à la réalisation de ce mémoire.

En fin, je remercie tous les membres de ma famille, mes amis pour leurs encouragements et leur contribution tout au long de mes études.

Résumé

Les protocoles TCP (protocoles de transmission fiable et en mode connecté) ont connu de nombreux modifications à travers diverses versions visant à améliorer non seulement la transmission de données entre deux processus en communication, mais également la gestion du réseau de façon générale.

Si TCP a connu et continue de connaître différentes versions avec des comportements différents dans des situations particulières, il apparait de plus en plus d'applications réseaux qui demandent un traitement approprié en termes de disponibilité du réseau. Ainsi, ces applications ne se contentent pas du service Best-effort du réseau, mais exigent une certaine bande passante garantie leur assurant une exécution optimale.

GTCP est l'un des rares protocoles TCP à pouvoir assurer ce service. Un autre protocole proposant le service garanti appelé TCP Xeno a été implémenté, par l'association de GTCP et de TCP Veno (pour une gestion efficace en cas de pertes).

Notre objectif est de valider le comportement de TCP Xeno, selon ses spécifications théoriques.

Pour cela, nous avons analysé les situations où TCP Xeno se comporte différemment des autres protocoles de contrôle de congestion. Des modules et des structures du kernel Lunix intervenant dans cette validation ont été décrits.

Abstract

The TCP (reliable transmission protocol and online mode) have undergone many changes through various versions to improve not only the transmission of data between two processes in communication, but also network management in general.

If TCP has been and continues to experience different versions with different behaviors in specific situations, more and more network applications appear that require proper treatment in terms of network availability. Thus, these applications are not only satisfied with the Best-effort network service, but require a certain bandwidth guarantee that ensures their optimal performance.

GTCP is one of the few TCP versions to be able to provide this service.

Another protocol providing guaranteed service is called TCP Xeno and has been implemented from the association of GTCP and TCP Veno (for effective management in case of loss).

Our goal is to validate the behavior of TCP Xeno, in certain situations as specified theoretically.

For this, we first conducted an analysis of situations where TCP Xeno behaves differently from other TCP protocols.

Modules and Linux kernel structures involved in this validation have been described.

Glossaire

ACK (Acknowledgement) il s'agit un paquet envoyé par le récepteur pour informr l'émetteur qu'il a bien reçu un paquet émis par ce dernier.

Best-effort Service sans garanties fourni par TCP aux applications pour amener les paquets à destination.

dupACK (Accusé de réception dupliqué) : Paquet envoyé par le récepteur à l'émetteur pour lui indiquer que le paquet reçu n'est pas celui qui était attendu. Le dupACK informe en même temps l'émetteur du numéro de séquence du paquet attendu par le récepteur.

MSS (Maximum Segment Size) : Taille maximale en nombre d'octet qu'un segment ne doit pas dépasser.

MTU (Maximum Transmission Unit) : Taille maximale en nombre d'octet qu'un frame (paquet au niveau de la couche liaison) ne doit pas dépasser

RTT (Round Trip Time) Temps nécessaire pour envoyer un petit paquet, de l'émetteur au récepteur et recevoir son accusé de réception.

TCP/IP Modèle représentant les protocoles utilisés par Internet en cinq couches (Application, transport, réseau, liaison de données et physique).

ssthresh: Variable définie par TCP et qui permet, lorsque la fenêtre de congestion atteint sa valeur, de passer de la phase Slow Start à celle de Congestion Avoidance.

Timeout : Expiration du temps accordé à un paquet pour qu'il soit transmis et acquitté.

Table des matières

Chapitre 1

1 Introduction générale

Depuis l'avènement du réseau Internet on s'efforce d'améliorer sans cesse l'efficacité de la transmission de données sur le réseau.

Cette amélioration passe entre autres par l'introduction de différents protocoles de transport.

Dès l'arrivée du réseau internet l'un des premiers soucis a été : Comment peut-on transmettre suffisamment de données dans un temps assez court ?

Ce défi consistant à augmenter la quantité de données transmises par unité de temps prend sens si on se rend compte du temps et des ressources (capacité de transmission) gaspillés par un protocole qui fonctionnerait en ne transmettant un paquet que lorsqu'il reçoit l'accusé de réception du paquet précédent (stop-and-wait protocol).

Si on estime le RTT entre deux terminaux A et B égal à 30 millisecondes, la taille de paquets à transmettre L égal à 8000 bits et le taux de transmission R de la ligne entre deux points A et B égal à 1Gbps $(10^9\ bits\ par\ seconde)$, un protocole de type stop-and-wait fournie à l'émetteur un lamentable taux d'utilisation de :

$$\frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$$

ce qui équivaut à un taux de transmission réel de 267 kbps, bien que l'on dispose d'une ligne dont le taux de transmission est de 1 Gbps [9, p. 214-215].

En utilisant plutôt un protocole capable de transmettre plusieurs paquets non acquittés à la fois, on arrive à améliorer considérablement la capacité de transmission de l'émetteur.

Cet exemple met en évidence combien un protocole de transmission est déterminant pour concrétiser les capacités des ressources disponibles d'un réseau.

Il existe deux grandes familles de protocoles de transport à savoir :

- TCP (un protocole de transport fiable, en mode connecté) et
- UDP (travaille en mode non-connecté et non fiable)

Dans l'optique d'amélioration du taux de transmission et de bonne gestion de ressources du réseau de façon générale, plusieurs versions de protocole TCP ont été implémentées, jusqu'au protocole TCP Xeno, objet de notre intérêt à travers notre sujet de mémoire qui consiste à la **validation du protocole TCP Xeno**. Cette validation consistera à décrire différents scénarios qui visent à prouver que TCP Xeno se comporte bien comme décrit dans ses spécifications.

Bien que le taux de transmission soit important, il ne suffit pas à lui seul pour garantir le fonctionnement efficace d'un réseau. En effet, la machine réceptrice doit être capable de suivre le rythme auquel le flux de données lui arrive, ou d'inviter l'émetteur à baisser son taux de transmission. Il s'agit d'un service de contrôle de flux (flow control) présent dans des protocoles TCP. Signalons que UDP ne fournit pas ce service.

A côte de ce service de contrôle de flux (côte récepteur) qu'un protocole de transmission de type TCP permet d'assurer, il y a également un service aussi important de contrôle de congestion.

Dans le deuxième chapitre nous présenterons les différentes caractéristiques d'un protocole de type TCP, ainsi que les principales fonctionnalités offertes par ce type de protocole.

Le troisième chapitre posera une analyse technique des différents types de protocoles TCP dont certains ont directement influencé l'implémentation du protocole TCP Xeno.

On remarquera que les apports d'une version à l'autre sont sauvent d'une importance capitale dans l'amélioration du taux de transmission entre deux hôtes en communication, et/ou de l'efficacité du réseau de façon générale.

Le quatrième chapitre nous permettra d'avoir un aperçu sur les différents modules et structures de données définies dans le Kernel Linux qui interviennent dans l'implémentation de protocoles TCP.

Le cinquième chapitre sera consacré à la mise en file d'attente de paquets sortant ou entrant dans le réseau. C'est dans ces files d'attentes que les opérations de filtrage de paquets seront effectuées.

Sur base de l'aperçu de modules et structures de données du quatrième chapitre et la présentation de mise en file d'attente du cinquième chapitre, le sixième et dernier chapitre est réservé à la description et l'intégration de l'évaluation de scenarios de validations dans l'implémentation du protocole TCP Xeno.

Le code du protocole TCP Xeno et des fonctions de validations seront présentés en annexe.

Chapitre 2

2 Protocole TCP

Les protocoles TCP se trouvent au niveau de la couche transport qui est la quatrième dans la pile protocolaire du modèle TCP/IP (Transport Control Protocol)/(Internet Protocol). Ils gèrent les communications de bout en bout entre processus.

La figure suivante représente la pile protocolaire du modèle TCP/IP :

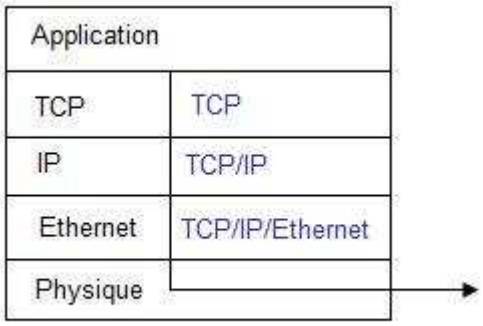

Figure 1 : Pile protocolaire du modèle TCP/IP

TCP est un protocole fiable orienté connexion qui assure le transfert de données (sous forme de flux d'octets) à destination dans l'ordre, sans altération, avec retransmission en cas de perte, et élimine les doublons.

TCP essaie de délivrer toutes les données correctement et en séquence. C'est son principal avantage sur UDP, même si ça peut être un désavantage pour des applications exigeant le transfert de flux en temps-réel, et pour lesquelles le critère temps prime sur la fiabilité.

UDP est généralement utilisé par des applications de type multimédia (audio et vidéo, …) pour lesquelles le temps qu'exige TCP pour gérer les retransmissions et l'ordonnancement des paquets n'est pas disponible. Ce genre d'application est tolérant par rapport à la fiabilité et/ou dispose des mécanismes d'extrapolation permettant de reconstituer d'éventuels paquets manquants.

TCP dispose d'un certain nombre de mécanismes pour assurer la fiabilité, l'ordonnancement de données, la détection de pertes et l'élimination des doublons.

En effet, l'entête d'un segment TCP est composé de champs contenant des informations nécessaires au protocole TCP pour garantir la fiabilité lors du transfert de données.

La figure ci-dessous représente la structure de l'entête d'un segment TCP :

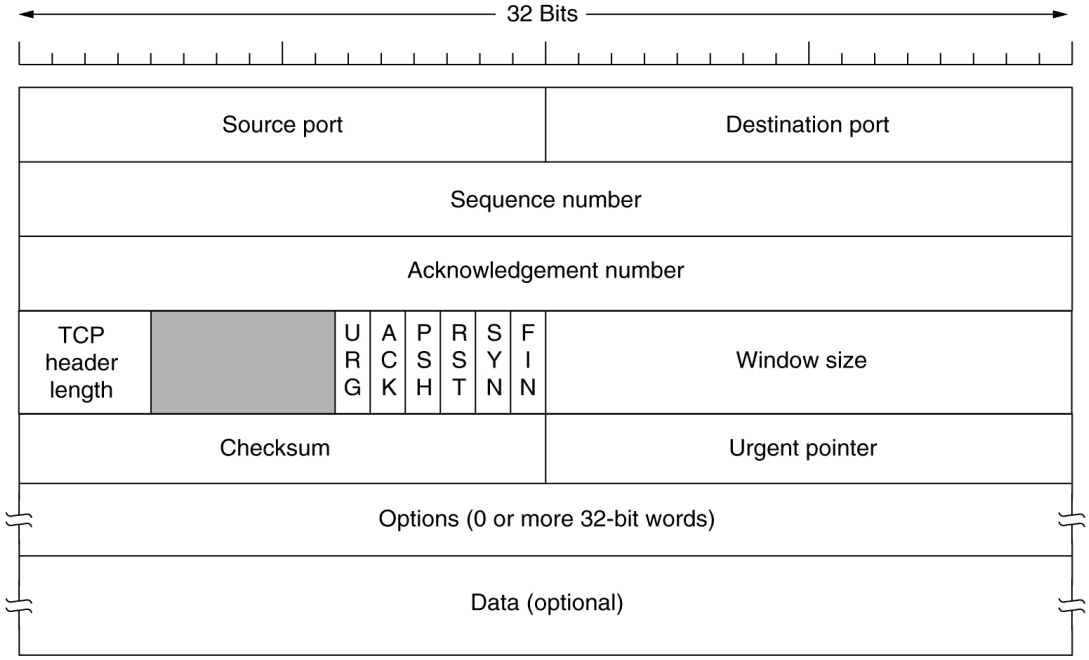

Figure 2 : Structure de l'entête d'un segment TCP [18]

Parmi les champs de l'entête de TCP, certains interviendront lors de la validation de TCP Xeno.

On peut citer :

- Numéro de séquence : Permet de détecter si les segments sont bien arrivés en ordre et les remettre en ordre sinon. L'ordre de segments peut être perturbé suite à une perte ou un retard de certains paquets dans le réseau. Les segments peuvent également être supprimés parce que le destinataire les a trouvés corrompus.

- Numéro d'accusé (Acknowledgment number) : Il informe l'émetteur du numéro de séquence du segment que le récepteur attend en fonction de segments déjà reçus.

- Internet checksum : L'émetteur remplit ce champ par une valeur obtenue en appliquant un algorithme de contrôle d'erreur au segment à envoyer. A l'arrivée, le récepteur fait de même au segment reçu et compare les deux valeurs pour s'assurer que le paquet n'a pas été altéré pendant son parcours dans le réseau. S'il arrive que le paquet ait été modifié, il est tout simplement supprimé.
  Seuls les segments en ordre et non corrompus sont envoyés à la couche supérieure (Application).

Lors d'une connexion TCP, la perte de données peut survenir soit dans le réseau (congestion), soit au niveau du tampon du récepteur si jamais le processus de l'application y tournant n'arrivait pas à consommer suffisamment de flux et provoquer le débordement du tampon.

Si TCP peut maîtriser le flux au niveau du récepteur, il ne peut que constater la congestion, par perte de données, sans possibilité de la prévenir. En effet, plusieurs connexions partageant le même réseau peuvent générer trop de trafic, qui une fois arrivé à un point d'étranglement (routeur à faible capacité de traitement ou une liaison à faible débit) engendrera une perte de données due au dépassement de la capacité du tampon au niveau du routeur.

Il existe cependant des protocoles TCP qui préviennent la perte de données par la détection du ralentissement d'une ligne congestionnée avant la survenance de perte de paquets. Nous en parlerons au chapitre trois.

Dans la partie qui suit nous allons décrire les mécanismes de contrôle de congestion et de flux de données mis en place par les protocoles TCP.

2.1 Contrôle de flux

TCP contrôle le flux en s'assurant que pendant la transmission de données entre deux hôtes, le tampon du côté récepteur ne soit jamais débordé par des segments en provenance de l'émetteur, ce qui induirait à une perte de segments en surplus.

Pour ce faire, soit :

rcvBuffer : La capacité du tampon du récepteur (en octets)

L : taille d'un segment

LastRead : Le dernier segment lu par le processus applicatif du côté récepteur

LastRCV : Le dernier segment reçu, la relation : $rcvBuffer \geq L(LastRCV - LastRead)$ doit toujours être vérifiée .

Le champ « receive window » dans l'entête d'un segment TCP annonce à l'émetteur, à chaque envoi d'ACK, la quantité de données que le récepteur est prêt à recevoir.

$L*receive\ window = rcvBuffer - L(LastRCV - LastRead)$

L'émetteur réagit en adaptant sa transmission de telle sorte que le nombre de segments envoyés et non encore acquittés ne dépasse pas l'espace du tampon disponible du côté récepteur.

Si on fait abstraction des éventuels segments renvoyés (out of order), l'émetteur doit vérifier que:

$LastSend - LastAck \leq receive\ window.$

Avec:

- LastSend : Le dernier segment envoyé
- LastAck : Le dernier segment acquitté

Ce qui revient à vérifier que: $(LastSend - LastAck) \leq rcvBuffer / L - (LastRCV + LastRead)$

Les deux figures suivantes représentent le mécanisme de contrôle de flux entre l'émetteur et le récepteur. La fenêtre glissante (côté émetteur) doit être régulée de telle sorte qu'à tout moment la fenêtre offerte soit inférieure ou égale à la partie libre du tampon (RcvWindow) du côté récepteur.

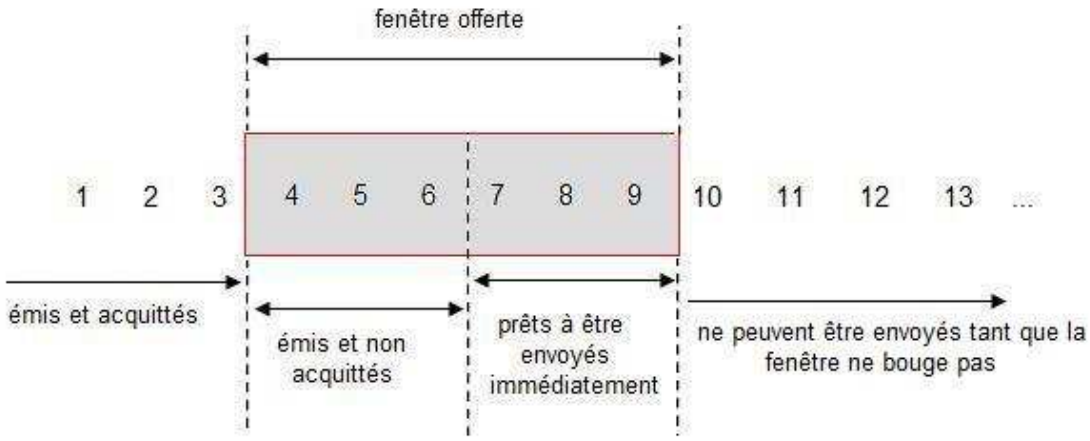

Figure 3 : Visualisation de la fenêtre glissante de TCP [19]

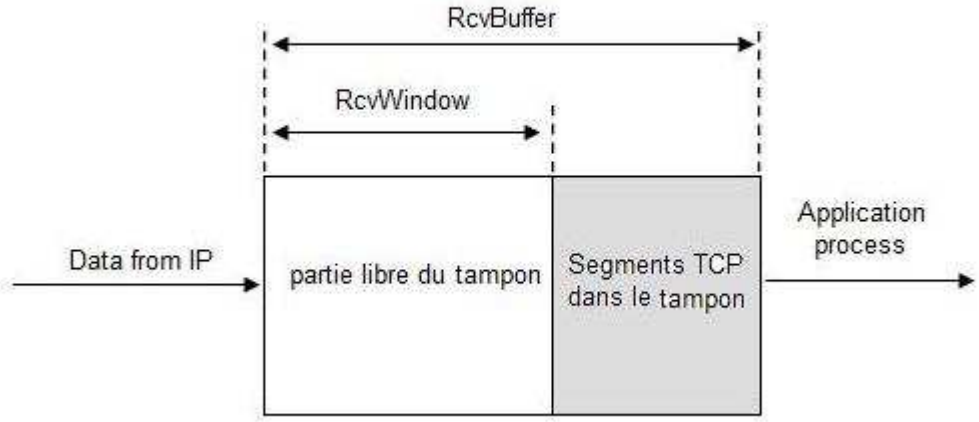

Figure4 : La fenêtre de réception et le tampon du coté récepteur [9]

Le contrôle de flux consiste à vérifier qu'à tout moment**: fenêtre offerte ≤ RcvWindow**

2.2 Contrôle de congestion

Comme on l'a déjà évoqué, le contrôle de flux vise à prévenir le dépassement de flux au niveau du tampon du récepteur. Rappelons que TCP est un protocole opérant de bout à bout (entre les processus en communication).

Quant à la congestion, TCP n'y a pas de contrôle aussi direct que pour le flux. En effet, la congestion survient dans le réseau au niveau des routeurs ou des liaisons qui ralentissent, suite à la présence d'une grande quantité de flux concurrents, en provenance et à destination d'une multitude d'hôtes.

En effet, plusieurs paquets de diverses provenances et différentes destinations peuvent se rencontrer au même nœud (routeur).

Il s'en suit que le taux de transmission dépendra de la vitesse de traitement de paquets par le routeur et du taux de transmission d'une liaison en aval, ce qui peut ralentir une liaison entre deux hôtes, et si le nombre de paquets affluant vers le routeur devient beaucoup plus important, le tampon du routeur est débordé et on assiste à une perte de paquets.

Ce service de contrôle de congestion assuré par un protocole de transport de type TCP, est ignoré par le protocole UDP dont le fonctionnement vise plutôt à fournir une grande vitesse de transmission sans se soucier d'éventuelles pertes.

Un autre service de plus en plus demandé aux protocoles de transport est sa capacité à pouvoir assurer une certaine bande passante aux applications plus exigeantes, plutôt que de les laisser se contenter du service « best effort ». Ce service est notamment proposé par le protocole TCP Xeno.

La figure ci-dessous représente une situation où une liaison de faible débit entre deux routeurs est susceptible de provoquer la congestion.

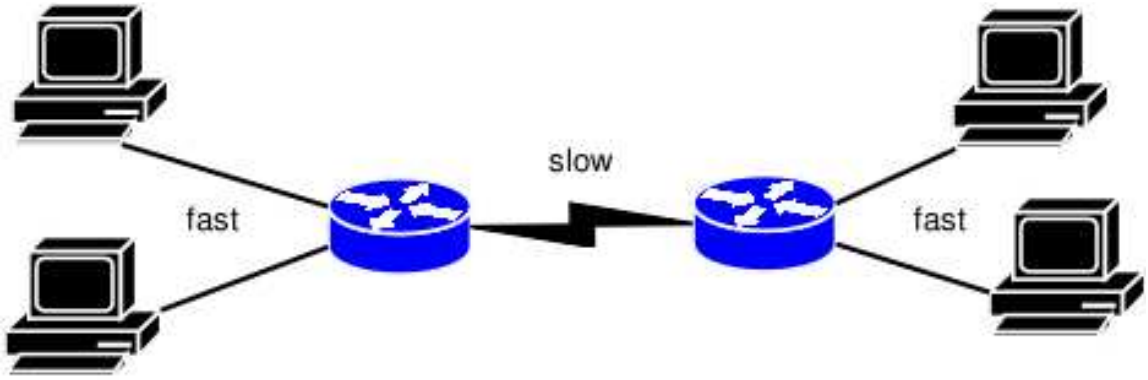

Figure 5 : Liaison qui ralentit entre deux routeurs [17]

TCP réagit aux symptômes de la congestion qui sont la perte de paquets ou le ralentissement de lien pour certaines versions de TCP que nous détaillerons dans le chapitre suivant.

L'émetteur adapte le débit des données envoyées à la bande passante réelle du réseau.

Notons que le contrôle de congestion est géré exclusivement par l'émetteur. Le récepteur ne fait que renvoyer des accusés de réception.

Il existe plusieurs mécanismes de gestion de trafic et de réaction à la congestion. Ces mécanismes comprennent l'utilisation d'algorithmes de démarrage lent (Slow Start), d'évitement de congestion (Congestion Avoidance), de retransmission rapide (Fast Retransmit) et de récupération rapide (Fast Recovery).

2.2.1 Démarrage lent (Slow Start)

L'algorithme Slow Start est utilisé au démarrage d'une connexion TCP ou après la survenance d'un timeout. Cette phase débute par une fenêtre de congestion limitée à un segment. A chaque arrivée d'un accusé de réception (après un temps égal à Round Trip Time), la fenêtre de congestion est augmentée d'une unité. Ce qui revient à doubler la fenêtre de congestion après chaque RTT.

Le but est de ramener l'émetteur à exploiter rapidement la bande passante disponible.

TCP passe de la phase Slow Start à la phase Congestion Avoidance lorsque la fenêtre de congestion atteint un certain seuil (threshold).

La phase Slow Start peut également être interrompue suite à la survenance d'un timeout. Dans ce cas, l'algorithme (Slow Start) recommence de nouveau.

2.2.2 Evitement de congestion (Congestion Avoidance)

Le protocole TCP passe en phase Congestion Avoidance lorsqu'il atteint le seuil (threshold). A ce moment précis la bande passante occupée par cette connexion TCP est jugée assez importante et le protocole devient moins agressif.

Plutôt que doubler la fenêtre de congestion (cwnd) après chaque RTT, elle est modestement augmentée de 1/cwnd à chaque réception d'un ACK (Additive Increase).

A l'apparition d'une perte (réception de 3 dupACK) ou à l'expiration d'un timeout , le seuil threshold est ramené à cwnd/2 .

La survenance d'un timeout est interprétée comme un signe d'une congestion sévère. La fenêtre de congestion est ramenée à 1, et on recommence par la phase slow start (MD pour Multiplicative-Decrease).

Signalons que le timeout est réinitialisé à chaque réception d'un ACK.

La figure ci-dessous représente les phases Slow Start et Congestion Avoidance suivies par l'apparition d'une perte.

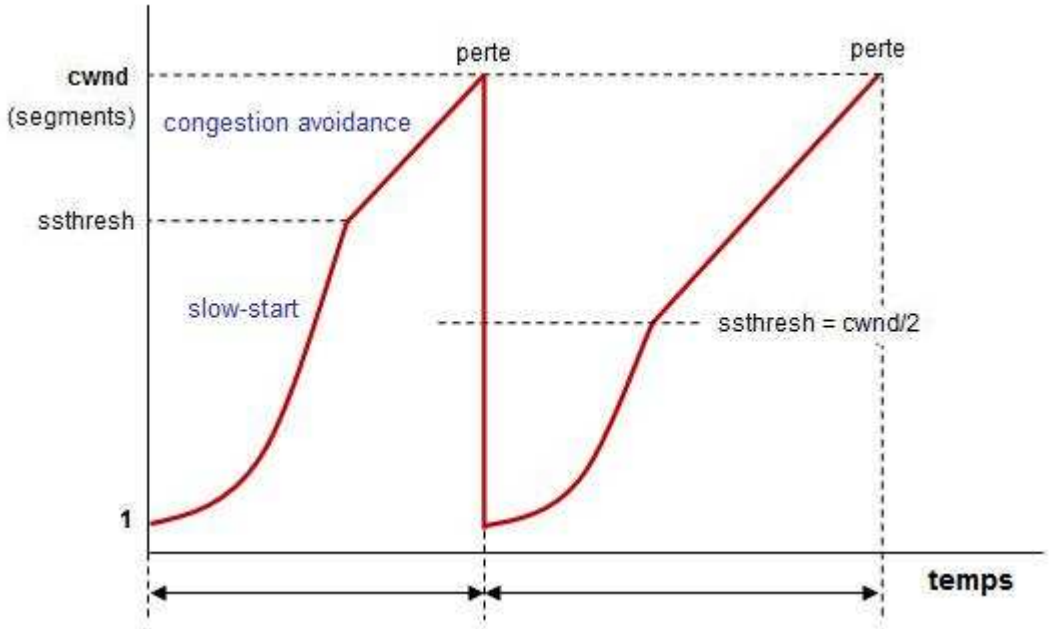

Figure 6: visualisation du Slow-Start et de Congestion Avoidance [3]

Lorsqu'il y a réception de 3dupACK, certains protocoles se comportent toujours comme s'il s'agissait d'un timeout.

Cependant avec la réception de 3dupACK, la plupart des protocoles jugent que la congestion n'est pas trop sévère étant donné que certains paquets arrivent à atteindre la destination, et se comportent différemment en réduisant de moitié la fenêtre de congestion et en repartant avec l'algorithme Congestion Avoidance (Additive-Increase).

Pour cela, des améliorations ont été proposées à l'algorithme de contrôle de congestion dont Fast Recovery et Fast Retransmit [20], [21], [22].

2.2.3 Retransmission rapide (Fast Retransmit)

La fiabilité des protocoles TCP exige que les segments soient reçus en ordre.

Ainsi, chaque accusé de réception annonce à l'émetteur le numéro de séquence du prochain segment attendu par le récepteur.

Lorsqu'il y a un segment qui n'est pas arrivé au récepteur (perte ou retard), à chaque réception d'un segment dont le numéro de séquence est supérieur à celui du segment attendu, le récepteur continue d'envoyer l'accusé de réception ayant le même numéro de séquence (du segment manquant).

A la réception du troisième accusé de réception (3dupACK) réclamant le même segment, l'émetteur considère le segment réclamé comme perdu et décide de le renvoyer sans attendre l'expiration du Timeout.

Par conséquent, on arrive à maintenir un nombre important de segments dans la connexion. Ce mécanisme est utilisé par TCP Reno [13].

Remarquons qu'à l'égard du récepteur, Il existe deux politiques de gestion de segments reçus en désordre :

- Soit ils sont tout simplement supprimés.
- Soit ils sont gardés dans un tampon en attendant la réception du segment manquant afin de les mettre en ordre et les transmettre au processus applicatif. Bien que cette seconde politique complique les fonctionnalités du récepteur, elle a le mérite d'épargner le réseau de la retransmission de paquets qui pourtant étaient bien reçus, et risqueraient à leur tour d'être perdus lors d'une nouvelle tentative.

2.2.4 Rétablissement rapide (Fast recovery)

Des protocoles de contrôle de congestion se comportent différemment selon qu'une perte est annoncée par l'expiration du Timeout (congestion sévère) ou la constatation de 3DupACK.

Pour le premier cas (l'expiration du Timeout):

- *Threshold = Cwnd/2*
- *Cwnd = MSS* (Maximum Segment Size)

Et, l'algorithme passe en mode Slow Start.

Pour le second cas (3DupACK):

La plupart de protocoles de contrôle de congestion passent en mode Fast Recovery.

En effet, bien qu'il y a eu perte, étant donné que la congestion n'est pas très sévère, on évite l'option radicale ci-dessus (cwnd = MSS et passage en mode Slow Start), pour ne pas priver à la connexion d'éventuelle bande passante disponible [23].

L'algorithme du Fast Recovery est caractérisé par :

- *Threshold = Cwnd/2*
- *Cwnd = Threshold + 3* ("gonflement" de cwnd) car il peut y avoir d'éventuels envois de nouveaux paquets pour chaque dupACK.
- Pour chaque dupACK, Cwnd++ , car il y a l'envoi éventuel d'un nouveau paquet
- Réception d'un non dupACK ("dégonflement" de cwnd) :
- *cwnd = ssthresh*
- retour au Congestion Avoidance

La figure ci-dessous représente le comportement d'un protocole TCP durant la phase de Fast Recovery. En comparant cette figure à la figure 4, représentant un protocole TCP passant par la phase Slow Start dès qu'une perte apparait, on remarque que Fast Recovery permet d'atteindre plus vite la bande passante disponible.

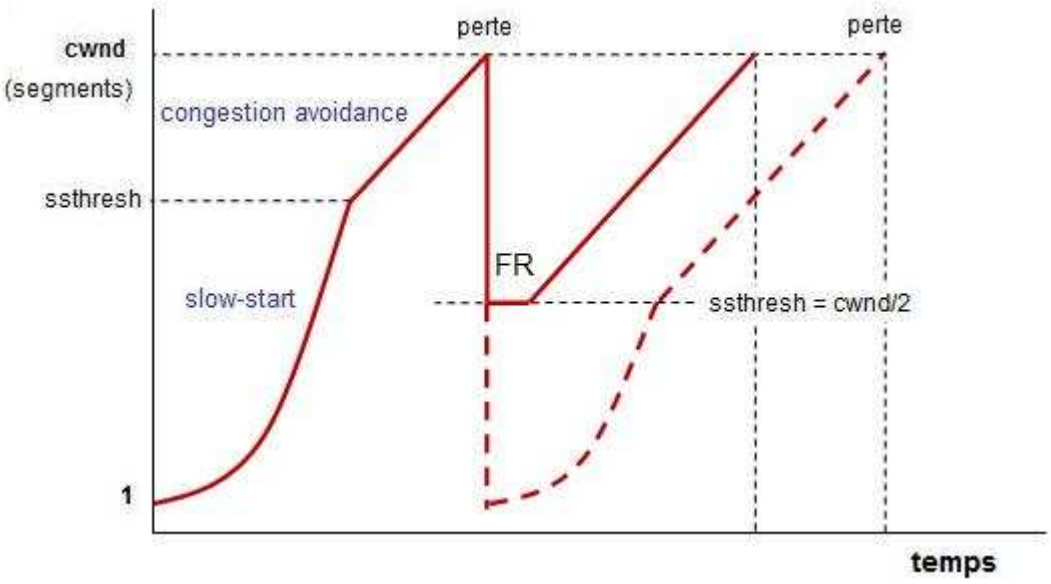

Figure 7: Visualisation du Fast Recovery [3]

Ce chapitre nous a permis de comprendre les différents mécanismes utilisés par des protocoles de contrôle de congestions TCP pour gérer le trafic dans le réseau. Le prochain chapitre va présenter comment différentes versions de protocole TCP ont adopté et adapté ces mécanismes dans leurs algorithmes de contrôle de congestion.

Chapitre 3

3 Quelques algorithmes de contrôle de congestion

Dans cette partie nous présentons les différentes versions de protocoles de contrôle de congestion dont certains (TCP Veno et TCP GTCP) ont directement contribué à l'élaboration du protocole TCP Xeno.

3.1 TCP Tahoe

C'est la plus ancienne version de TCP. Il est caractérisé par trois phases successives :

- Slow-Start
- Congestion Avoidance
- Fast Retransmit

Dès qu'il détecte une perte, soit par la réception de 3 dupACK, soit par l'expiration d'un timeout, il réduit sa fenêtre de congestion à 1MSS et passe en phase Slow Start (Fast Retransmit).

L'initialisation commence avec la phase Slow Start. La taille de la fenêtre de congestion est fixé à 1 (MSS).

Pendant cette phase la fenêtre de congestion (cwnd) croit de façon exponentielle. A chaque réception d' ACK, *cwnd = cwnd +1*.

TCP Tahoe quitte cette phase pour celle de Congestion Avoidance lorsque cwnd devient supérieur à ssthresh.

Durant la phase de Congestion Avoidance la taille de la fenêtre croît linéairement de 1 MSS après la réception d'ACK pour tous les segments de la fenêtre courante.

3.2 TCP Reno

La plupart des implémentations TCP utilisent actuellement l'algorithme de Reno.

En plus des phases de Tahoe dont Fast Retransmit, TCP Reno dispose de la phase de Fast Recovery décrite au chapitre 2, et caractérisée entre autres par l'augmentation de la fenêtre par la taille d'un paquet à chaque acquittement dupliqué reçu.

Contrairement à Tahoe, à la détection de perte par la réception de 3 dupACK, seul le paquet supposé être perdu est retransmis et, Reno ne passe pas en Slow-Start. Sa fenêtre de congestion devient (cwnd = ½ cwnd + 3), le ssthresh est ramené également à ½ cwnd et, continue en phase de Congestion Avoidance avec la nouvelle fenêtre de congestion et le nouveau ssthresh.

On peut facilement constater que Reno est meilleur que Tahoe, cependant ce n'est pas toujours le cas. En effet, lorsque surviennent plusieurs pertes non consécutives dans une même fenêtre, les performances de Reno se dégradent de manière drastique.
Cela est dû au fait que la fenêtre de congestion peut décroître plusieurs fois dans un même RTT.

Une nouvelle version nommée **TCP Newreno** [24] propose une légère modification de la phase Fast Recovery de Reno, pour remédier à ce problème.

TCP Newreno reste en Fast Recovery jusqu'à la réception des accusés de tous les paquets perdus dans la rafale de paquets initiale. D'où son efficacité, car il coupe cwnd seulement après la première perte.

3.3 TCP Vegas

En 1994, **Brakmo, O'Malley et Peterson** ont présenté une nouvelle implémentation de TCP appelée TCP Vegas qui, selon ses auteurs est capable d'interagir avec toute implémentation valide de TCP. Ce protocole permet également d'améliorer de 37% à 71% le débit par rapport à Reno, et d'éviter entre un cinquième et la moitié de pertes que connaîtrait Reno [13], [14].

L'algorithme de contrôle de congestion proposé par TCP Vegas essaye d'éviter la congestion en maintenant un bon débit dans le réseau.

Vegas détecte la congestion par la variation de la valeur du RTT (plus RTT est long, plus il y a de la congestion dans le réseau) et adapte le débit avant que ne survienne la perte de paquets.

Son mécanisme de fonctionnement se présente comme suit :

Slow-Start

Le mécanisme de Reno consistant à doubler la taille de la fenêtre de congestion à chaque RTT au cours de la phase Slow-Start est très coûteux en termes de pertes car, une fois la bande passante disponible atteinte, on peut enregistrer une perte de la moitié de la fenêtre de congestion.

TCP Vegas essaie de tester constamment la bande passante disponible pour éviter ce genre de pertes.

Pour ce faire, Vegas double la taille de sa fenêtre seulement tous les deux RTT, contrairement à chaque RTT dans le cas de Reno.

Pendant ce temps, deux taux ou débits (Expected et Actual) sont calculés :

$$Expected = cwnd/BaseRTT \text{ et}$$

$$Actual = cwnd/RTT$$

Où :

- Expected est le débit maximum
- Actual est le débit réel de transmission

- cwnd est la taille de la fenêtre de congestion

- BaseRTT est la valeur minimale de tous les RTT mesurés

- RTT est le temps aller-retour d'un paquet particulier.

Si la différence entre *Expected* et *Actual* dépasse un certain seuil γ, Vegas bascule de Slow Start à Congestion Avoidance.

Congestion Avoidance

Durant cette phase les débits *Expected* et *Actual* sont calculés comme on l'a vu ci-dessus.

Posons *Diff = Expected – Actual*

On remarque que Diff ≥ 0 . Si *Actual ≥ Expected* , cela implique qu'on doit changer BaseRTT par le plus récent RTT.

L'émetteur définit deux seuils a et b avec a < b qui correspondent, respectivement à la présence de petite et de grande quantité de données stockées dans la file d'attente intermédiaire.

De tel sorte que si Diff < a, Vegas augmente linéairement sa fenêtre de congestion au cours du prochain RTT.

Si Diff > b, Vegas diminue linéairement sa fenêtre de congestion au cours du prochain RTT.

Et lorsque a < Diff < b, la fenêtre de congestion reste inchangée.

On constate que TCP Vegas peut facilement souffrir de la concurrence des protocoles comme Reno, qui continuent à augmenter leur fenêtre et ne réagissent à la congestion que lorsqu'apparaît une perte.

3.4 TCP Veno

TCP Veno est un protocole de contrôle de congestion adapté à des réseaux où la perte aléatoire de paquets est considérable (réseaux sans fils).

Veno suit l'état de congestion du réseau pour qu'en cas de perte il puisse décider si la perte est plus probablement due à la congestion ou à une éventuelle erreur de bit dans le paquet.

Dans un réseau sans fil soumis à une perte aléatoire de 1%, l'utilisation de TCP Veno peut améliorer le rendement jusqu'à 80% [1].

Veno utilise des mécanismes similaires à ceux de Vegas pour estimer l'état d'une connexion. Les informations reçues par Veno sur l'état de la connexion servent cependant en cas de perte, pour évaluer la cause de cette perte (perte due à la congestion ou perte aléatoire), contrairement à Vegas qui les utilise pour essayer d'éviter la survenance de pertes.

Pour cela, Veno mesure aussi deux taux : *Expected* et *Actual.*

$$Expected = cwnd/BaseRTT$$

$$Actual = cwnd/RTT$$

Soit Diff = $Expected - Actual$

Si N est le nombre de paquets accumulés dans la file d'attente du lien qui ralentit on a :

$$RTT = BaseRTT + N/Actual$$

En réarrangeant la formule on a :

$$N = Actual * (RTT - BaseRTT) = Diff * BaseRTT$$

Lorsqu'une perte est détectée, en fonction de N, Veno décide s'il s'agit d'une perte due à la congestion du réseau ou s'il s'agit d'une perte aléatoire (modification de bit d'un paquet).

Si $N > \beta$ au moment de la perte, Veno considère cette perte comme due à la congestion et adopte l'algorithme de Reno pour ajuster sa fenêtre. Par contre si $N < \beta$, la perte est aléatoire et la fenêtre de congestion est ajustée différemment de la façon dont le fait Reno.

Par expérience β est fixé à 3.

3.4.1 Comparaison par rapport aux algorithmes fondamentaux de Reno

Slow Start

Durant la phase de Slow Start TCP Veno se comporte exactement de la même façon que Reno. A chaque accusé de réception, la fenêtre de congestion est augmentée d'une unité.

Algorithme Additive increase

TCP Veno modifie l'algorithme Additive Increase de Reno de manière suivante:

Si ($N < \beta$), cela signifie qu'il existe une bande passante encore disponible, et Veno garde le comportement de Reno.

La réception de chaque accusé de réception s'accompagne de l'augmentation de la fenêtre de congestion (de cwnd à cwnd + 1/ cwnd).

Par contre, si ($N \geq \beta$), c-à-dire que la bande passante est totalement utilisée, TCP Veno adopte un comportement intermédiaire entre celui de TCP Vegas (diminution de la fenêtre de congestion de façon préventive) et de Reno (continuer à augmenter cwnd de 1/cwnd, à chaque réception d'un nouveau ACK).

En effet, TCP Veno augmente sa fenêtre de congestion cwnd de 1/cwnd, non pas à chaque réception d'ACK, mais une fois par réception de deux nouveaux ACK. Cela a comme avantage, étant donné que la bande passante est totalement utilisée, de garder une grande fenêtre de congestion plus longtemps (en évitant une perte due à la congestion) que Reno, car elle croît moins rapidement lorsque la région critique ($N \geq \beta$) est atteinte.

Algorithme Multiplicative Decrease

Avec Reno, une perte est détectée de deux façons (expiration du timeout et réception de 3 dupACK) :

- Lorsqu'un paquet n'est pas acquitté à l'expiration du timeout, la phase Slow Start est lancée, avec ssthresh ramené à cwnd/2 et cwnd ramenée à 1. L'expiration du timeout est un signe que la congestion est sévère, et le taux de transmission est réduit drastiquement.

  TCP Veno ne modifie pas cette partie de l'algorithme.

- A la réception de 3 dupACK, Reno utilise le mécanisme Fast Retransmit suivie de Fast Recovery. TCP Veno modifie la première partie de l' algorithme Fast Recovery en [1]:

```
If (N < β) random loss due to bit errors is most likely to
                                                have occurred
    ssthresh = cwnd * (4/5);
  else    // congestive loss is most likely to have occurred
          ssthresh = cwnd/2 ;
```

On remarque que Veno fait une distinction entre une perte due probablement à une erreur de bit et une perte due à la congestion.

Pour la première perte, Veno diminue la taille de la fenêtre de congestion et du ssthresh d'un cinquième au lieu de la moitié de la taille atteinte au moment de la détection de la perte comme c'est le cas, en cas de congestion.

La figure ci-dessous montre le comportement de TCP Veno durant la transmission de segments.

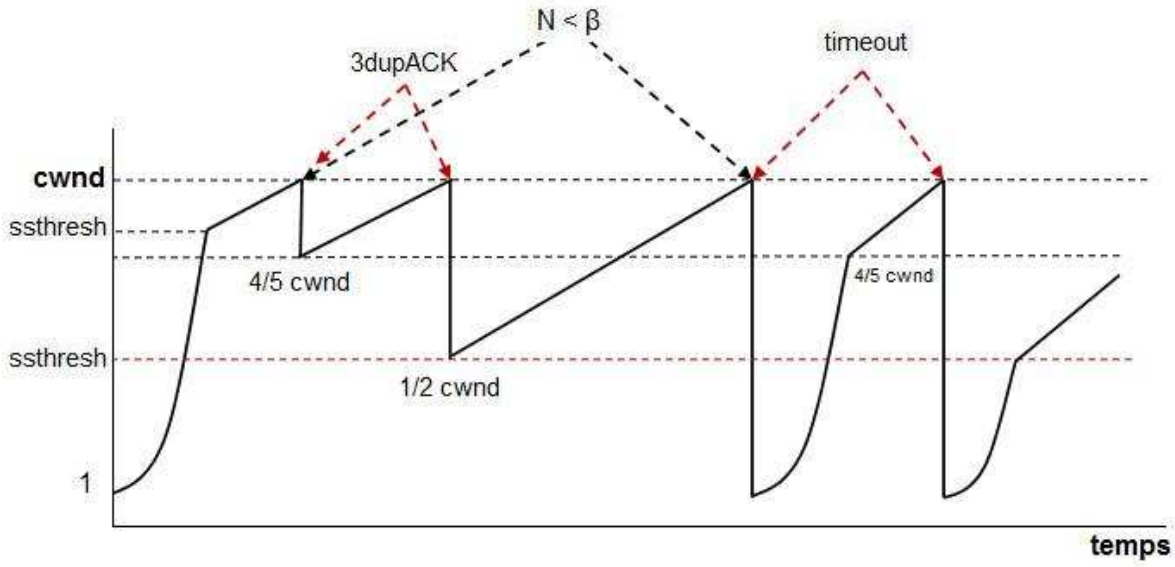

Figure 8 : Comportement de TCP Veno [3]

3.5 GTCP

GTCP est adapté aux applications exigeantes en bande passante garantie, qui ne peuvent pas se contenter du service best-effort fourni par un protocole TCP n'offrant pas de possibilité de réservation de bande passante.

La composante garantie est définie par la relation :

$cwnd_G = rate_G * rtt_{base}$

avec $rate_G$, taux du service garantie et rtt_{base} , le minimum des RTT qu'a connu la connexion jusque là.

$$rtt_{base} = min\ (rtt_{base}, rttsample)$$

En plus de la composante garantie, ces applications bénéficient d'une autre composante best-effort comme n'importe quelle autre application.

La fenêtre de congestion devient : $cwnd = cwnd_{G+}cwnd_{BE}$

Où $cwnd_{BE}$, est la composante best effort dont la gestion ne tient pas compte de l'existence de la composante garantie.

L'adoption de ce protocole nécessite quelques changements du côté émetteur (le récepteur reste avec le fonctionnement d'un TCP normal).

Phase Slow Start du protocole GTCP

L'algorithme de Slow Start utilisé par GTCP au démarrage est différent de celui qu'il utilise après l'apparition d'un timeout.

- Au démarrage, l'algorithme Slow Start est appliqué aux deux composantes (garantie et best effort) : cwnd = 2

  A la réception d'un ACK, la fenêtre de congestion est incrémentée (cwnd++)

GTCP quitte la phase Slow Start lorsque la fenêtre de congestion devient plus grande que la somme de ssthresh et de la valeur idéale de la composante garantie de la fenêtre de congestion ($rate_G$ * rtt_{base}).

- A la survenance d'un timeout, la fenêtre de congestion est ramenée à 1. Le plus grand numéro de séquence de paquet envoyé est enregistré dans une variable (ssexitthresh = maximum segment number).

  C'est lorsque le bord gauche de la fenêtre de congestion dépasse ssexitthresh que GTCP quitte la phase Slow Start et ramène sa fenêtre de congestion à $cwnd_G + cwnd_{BE}$.

On remarque que l'algorithme Slow Start pour un Timeout Recovery n'est appliqué qu'à la composante best-effort. La composante garantie est maintenue à son niveau de réservation. Ainsi, le comportement de GTCP au démarrage est différent de celui qu'il affiche à l'apparition d'un timeout.

Congestion Avoidance

Durant cette phase GTCP utilise le mécanisme de TCP. A la réception de chaque ACK, cwnd est incrémentée de $\frac{1}{\text{cwnd}}$.

Quand une perte est détectée avec la réception de trois dupACK, seul la composante best effort est divisée par deux. La fenêtre de congestion devient :

$$cwnd = cwndG + \frac{cwndBE}{2}$$

Fast Recovery

Rappelons que Fast recovery est un mécanisme permettant aux protocoles TCP de continuer à transmettre les données pendant la récupération de données.

En cas de k pertes et si la fenêtre de congestion est cwnd :

Le nombre idéal de paquets à envoyer à chaque réception d'ACK est:

$$FRi_{deal} = max\left(cwndG,\ \ cwnd - k - \frac{cwndBE}{2}\right)$$

On remarque cependant que si le nombre k de pertes est important et devient supérieur à $\frac{\text{cwndBE}}{2}$ TCP ne pourra pas envoyer les données désirées.

On a :

$$FRi_{TCP} = cwnd - k - \frac{cwndBE}{2}$$

FRi_{TCP} peut être inférieur à cwndG si k > ½ cwndBE. Pour résoudre ce problème, GTCP utilise un mécanisme de Fast Recovery modifié décrit ci-dessous [2]:

- Lorsqu'une perte est détectée GTCP enregistre la fenêtre de congestion mise à jour

$$cwnd_{update} = cwndG + \frac{cwndBE}{2}$$

- GTCP, effectue une transmission en quelque sorte forcée pour les premiers cwndG dupACK qui suivent immédiatement la détection de perte, et augmente cwnd à chaque transmission.

- Après la transmission de cwndG, les ½ cwndBE (ou les cwndBE – k, si k > ½ cwndBE) dupACK sont ignorés, et cwnd reste inchangée.

- GTCP transmet de nouveaux segments pour les éventuels dupACK suivant ½ cwndBE (ou les cwndBE – k, si k > ½ cwndBE) dupACK

- Quand l'ACK pour le numéro maximal de segment au moment de la détection de la perte arrive (FullACK), GTCP ramène sa fenêtre de congestion à $cwnd_{update}$.

La figure ci-après représente le comportement du protocole GTCP

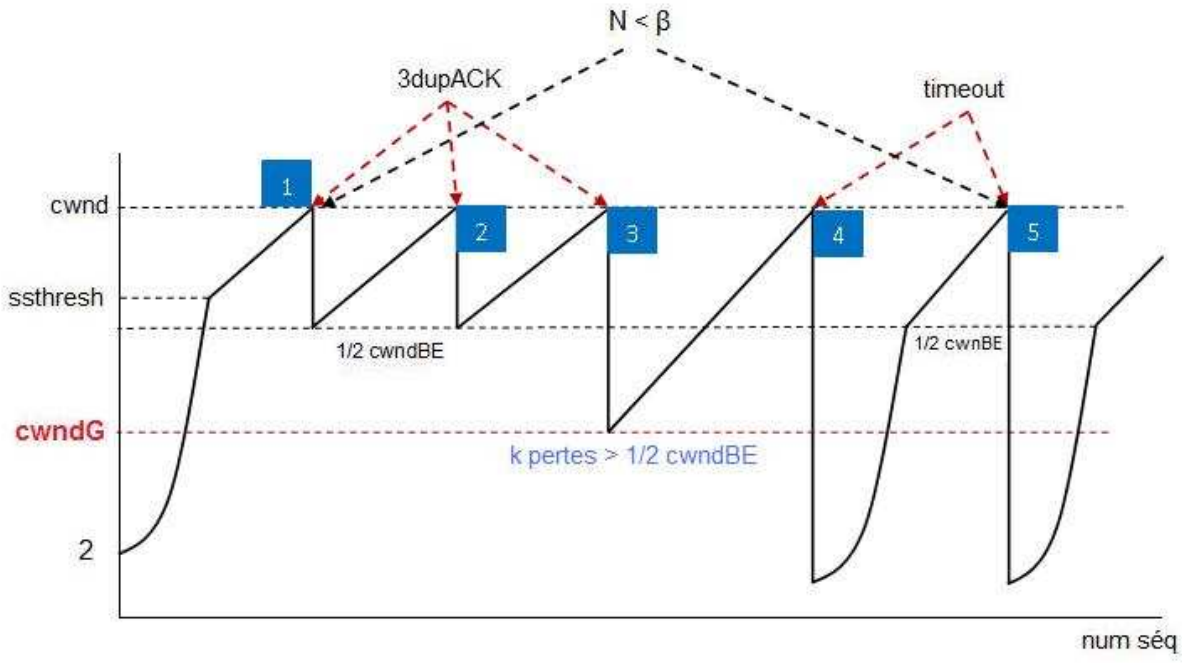

Figure 9: Comportement du protocole GTCP [3]

3.6 Protocole de contrôle de transmissions TCP_Xeno

3.6.1 Introduction

Le protocole **TCP Xeno** résulte de l'interaction entre les protocoles GTCP [2] et TCP Veno [1].

GTCP a le mérite de permettre de garantir aux applications qui l'utilisent une certaine bande passante, en plus du service « best-effort » fourni par le réseau. Ainsi GTCP est adapté à un environnement où certaines applications exigent un débit garanti pour une exécution normale. Il s'agit d'applications genre vidéoconférence, « streaming live audio/video », la téléphonie sur internet, … où l'interaction entre différents acteurs doit se faire en temps réel.

Lorsqu'une situation où le débit réservé ne peut pas être garanti par le réseau, une bonne gestion consisterait à interrompre l'exécution de l'application exigeante plutôt que de fournir un service médiocre, et en même temps gaspiller la bande passante qui aurait pu être utile à d'autres applications moins exigeantes qui se contentent du service best effort.

Cependant, en cas de perte GTCP applique l'algorithme « **Multiplicative Decrease (MD)**», destiné à limiter la congestion dans les différents nœuds du réseau, et passe en mode « Fast Recovery », sans vérification préalable de causes à l'origine de cette perte.

Or, un réseau internet peut être filaire ou sans fil. Dans le second cas, une perte de paquets ne signifie pas nécessairement qu'il y a congestion. Dans un réseau sans fil une perte peut résulter d'une collision, mauvaise liaison (obstacle entre l'émetteur et le récepteur), différents bruits…

En confondant la perte de paquets avec l'apparition de congestion dans le réseau, non seulement on prive l'application utilisant GTCP de la bande passante disponible, mais également, elle est désavantagée par rapport à d'éventuelles applications (genre multimédia à temps réel sur UDP. Cela provoque une utilisation non optimale du réseau. D'où l'idée d'améliorer GTCP avec une capacité à différencier les pertes.

Quant à TCP Veno, protocole ne supportant pas des réservations de largeur de bande, l'apparition de pertes de données ne conduit pas à une conclusion immédiate qu'il y a congestion du réseau, mais permet de vérifier l'état du réseau en comparant N et *beta* (β), avec N, le nombre des paquets accumulés dans la pile d'attente du lien qui ralentit, au moment de la détection de pertes [1], et β un paramètre de seuil fixé à 3 sur foi d'expériences, comme on l'a déjà vu à la section 3.4.

Si $N < \beta$, Veno considère que la congestion n'est pas sévère et se comporte différemment de (G)TCP.

Par contre si N > β, dans ce cas la congestion est jugée sévère et Veno garde le comportement de (G)TCP.

Lors de son mémoire Mr Christos Styliaras [3] a intégré les fonctionnalités de GTCP et celles de TCP Veno pour élaborer un nouveau protocole nommé TCP Xeno.

TCP Xeno permet la réservation de largeur de bande à la manière de GTCP, et il gère intelligemment les cas de perte comme TCP Veno.

Un autre avantage de TCP Xeno est que son adoption ne nécessite d'intervention que du côté émetteur, par la compilation, l'installation et le chargement de son module, sans autre modification ni au niveau des routeurs ni au niveau du récepteur.

Dans le chapitre 2 nous avons énoncé quelques caractéristiques et mécanismes de fonctionnement de protocole TCP en vue d'assurer la régulation du trafic dans le réseau. Ces caractéristiques nous ont permis de présenter le comportement de quelques versions de protocoles TCP dans ce troisième chapitre.

Nous remarquons qu'au fil de l'histoire des réseaux, les protocoles TCP ont dû être améliorés pour mieux répondre soit aux situations réelles des réseaux, soit à des exigences de certaines applications qui demandent un traitement particulier.

Dans le chapitre suivant, on verra comment un paquet est représenté dans le kernel Lunix. Pour cela, on commencera par identifier les modules impliqués dans la gestion du réseau au niveau du kernel, ainsi que des structures de données **sock et sk_buff** qui contiennent des informations respectivement, sur l'état du réseau, et représentant des paquets envoyés ou reçus.

Chapitre 4

4 Gestion du réseau dans le Kernel Linux.

Le kernel de Linux joue un grand rôle dans le système. Il est l'intermédiaire (interface) entre les différents programmes et le matériel, il gère de la mémoire pour les programmes en cours d'exécution et s'assure que le processeur soit bien partagé.

Dans le kernel de Linux, les modules impliqués dans la gestion du réseau se trouvent dans les répertoires suivants:

/usr/src/linux.2.6.16.23/net/ipv4

/usr/src/linux.2.6.16.23/net/core

/usr/src/linux.2.6.16.23/net/sched

/usr/src/linux.2.6.16.23/include/linux

/usr/src/linux.2.6.16.23/include/net

Les différents types de protocoles TCP, IP, Ethernet, … utilisent, à leurs niveaux respectifs, des fonctions, des structures de données, des variables, … définies dans ces modules pour implémenter leurs algorithmes de gestion du réseau.

La figure suivante représente sous forme d'arborescence certains répertoires du kernel Linux. Les répertoires qui sont impliqués dans la gestion du réseau sont représentés en gras.

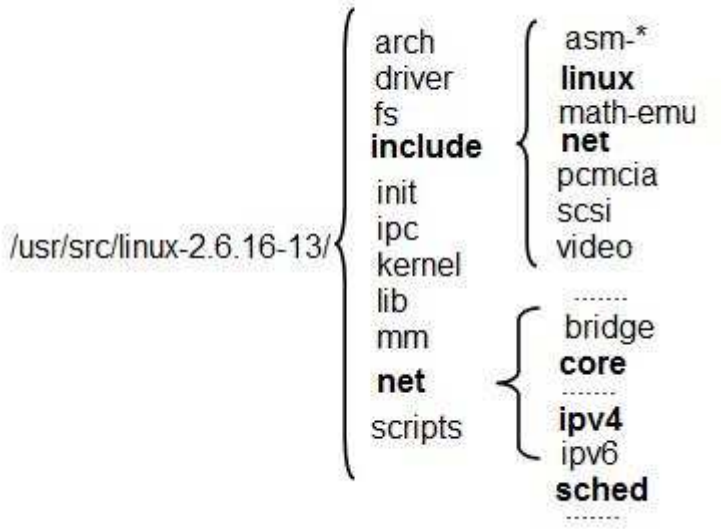

Figure 10: Networking code in the Linux kernel tree [4]

4.1 TCP dans le kernel linux

Des protocoles de gestion de la couche transport (TCP) se servent de ces définitions pour implémenter leurs algorithmes de gestion de la congestion et de gestion de flux de données entre l'émetteur et le récepteur.

Ils utilisent un certain nombre de structures de données dont les plus importantes sont sock et sk_buff. Ces structures contiennent des variables (membres) utilisées dans les algorithmes de gestion de la congestion. Dans la suite, on décrira le rôle de ces structures.

4.1.1 La structure sock

sock est une structure définie dans le répertoire /include/net/sock.h du kernel de Linux dont les membres contiennent des informations sur l'état d'une connexion. La structure sock est allouée à chaque fois qu'un socket est créé [4] afin d'établir une connexion entre deux machines hôtes.

Les premiers champs indiquent des adresses IP source et destination et des numéros de port source et destination.

Parmi les champs (membres), certains contiennent des informations propres à chacune des couches protocolaires d'un paquet (transport, réseau, liaison de données).

Le champ de type structure tcp_opt est particulièrement intéressant pour la validation du protocole TCP Xeno, puisqu'il contient des informations utilisées par les protocoles TCP. Signalons que les protocoles TCP sont orientées connexion ce qui explique une grande quantité de données que doit garder la structure tcp_opt, contrairement aux structures liées aux protocoles IP, UDP, qui eux contiennent relativement moins d'information.

Parmi les champs que contient la structure tcp_opt, les plus pertinents sont :

- tcp_header_len : nombre d'octets dans l'entête TCP du paquet à envoyer
- snd_nxt : le numéro de séquence du paquet à envoyer
- snd_una : premier octet pour lequel on attend un accusé de réception

4.1.2 La structure sk_buff

Cette structure est utilisée pour stocker des informations relatives aux entêtes de différentes couches (TCP, IP, Ethernet,…), les informations en provenance de la couche application (charge utile pour un segment TCP), ainsi que d'autres informations nécessaires à la gestion du réseau.

Lorsqu'un paquet sortant ou entrant arrive au kernel Linux une structure de type sk_buff liée à ce paquet est crée.

Toute modification d'un champ d'un paquet doit se faire par le biais du champ correspondant de cette structure.

4.1.2.1 Quelques champs de la structure sk_buff

- Les sk_buff se présentent dans le kernel Linux comme une liste doublement chaînée, chacun pointant respectivement vers son prédécesseur et son successeur, à l'aide de champs pointeurs **sk_buff *prev** et **sk_buff *next**.

- Une structure **head *list** pointe vers l'entête de la liste à laquelle appartient un sk_buff.

- Un pointeur ***sk**, est réservé pour une structure sock à laquelle le sk_buff est attaché (cas d'un sk_buff représentant un paquet sortant) ou sera attaché (cas d'un sk_buff représentant un paquet entrant).

- Une structure **timeval stamp** indique à quel moment un paquet est arrivé (moment auquel une structure sk_buff correspondant à un paquet a été créée).

- Le pointeur ***dev** enregistre l'interface par lequel un paquet est arrivé. Pour un paquet sortant le champ est rempli si l'interface est connu, en consultant une table de routage par exemple [4].

Les champs suivants nous intéressent particulièrement, car ils correspondent aux structures d'entête de paquet à différents niveaux des couches protocolaires. On rappelle que c'est dans un champ de l'entête TCP/IP que, lorsque les conditions de réalisation d'un scenario parmi les scenarios à tester seront remplies, les modifications seront apportées afin que le paquet soit filtré (reconnu) et supprimé par Netem.

- Le champ correspondant à l'entête de couche transport est une union « **h »** de structure pointant vers des entêtes de divers types de couches transport (TCP, UDP, …)

- Le champ pour l'entête de la couche réseau est une union « **nh »** de structure pointant vers différent types d'entêtes de la couche réseau (IPV4, IPV6, …)

- Le champ pour la couche liaison de données est aussi une union « **mac** » de deux pointeurs de structure, l'un (le plus utilisé) pointant vers l'entête d'une trame de technologie Ethernet.

On remarque que d'après les différents champs de la structure **sk_buff**, surtout les champs liés à l'entête des paquets, le marquage d'un paquet peut se faire aussi bien dans son

entête IP que dans son entête TCP. En effet, les deux structures d'entête sont accessibles à partir de la structure sk_buff correspondant à un paquet entrant ou sortant du kernel Linux.

Nous choisirons d'utiliser le champ TOS de l'entête IP, dont les bits seront modifiés à 11111111 pour un paquet à supprimer. En effet, bien qu'il soit possible de classer et filtrer les paquets sur base du contenu de leurs entêtes TCP, l'utilisation de l'entête IP est plus simple et directe comme on le verra dans la partie réservée à cet effet.

La figure ci-dessous montre la structure sk_buff et sa relation avec d'autres structures

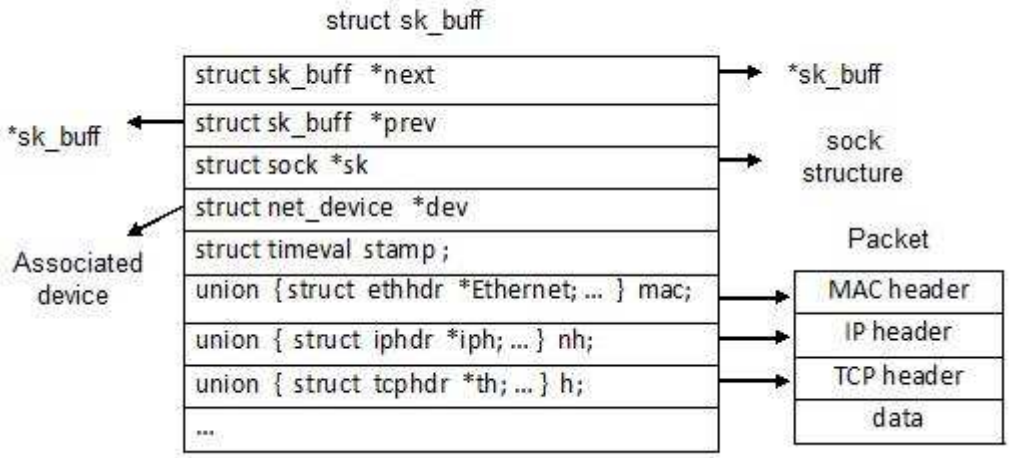

Figure 11: structure sk_buff [5]

On ne va pas énumérer tous les champs composant une structure sk_buff. Une description plus détaillée peut être trouvée dans [4].

4.2 Implémentation de TCP Xeno

4.2.1 Structure de Hemminger

Dans l'implémentation du module TCP Xeno, on a utilisé la structure de codage définie par *Stephen Hemminger* qui fournie une interface commune pour les algorithmes de contrôle de congestion [25], [26].

L'interface indique les différentes fonctions (qui détectent l'état du réseau et réagissent en conséquence) qu'on peut implémenter. Quatre fonctions sont obligatoire à savoir : init (), ssthresh (), min_cwnd () et cong_avoid ().

Le but étant d'éviter de devoir modifier le code du kernel Linux, lors de l'implémentation de nouvelles versions de protocoles TCP.

Ainsi, chaque fonction définie dans le module tcp_xeno est d'une certaine manière indépendante des autres à part la fonction *tcp_xeno_rtt_init()* qui est appelée dans la fonction *tcp_xeno_cwnd_event().* Signalons que la fonction *tcp_xeno_rtt_init()* est définie localement dans l'implémentation de TCP Xeno et ne fait pas partie de fonctions fournie par l'interface définie par *Stephen Hemminger* .

4.2.2 Constatations

- La structure de codage définie par *Stephen Hemminger* facilite l'implémentation de nouveaux modules TCP et le chargement/déchargement de différents modules, mais en même temps elle rend cette implémentation très abstraite (l'interaction entre les différentes fonctions est cachée).

- La plupart de fonctions sont appelées à la réception d'un ACK et non à l'envoie d'un segment. Cela peut se justifier par le fait que l'information intéressante du point de vue de l'émetteur pour ajuster sa transmission, est celle en provenance du récepteur contenue dans l'accusé de réception.

Le chapitre 5 s'intéressera sur la façon dont des paquets sortants sont filtrés et certains supprimés sur base de critères précis.

Chapitre 5

5 Gestion de la mise en file d'attente de paquets

Une fois les paquets produits, ils passent par un mécanisme de mise en file d'attente (Queues) associée à l'interface de sortie. Il peut y avoir également une mise en file d'attente associée à l'interface d'entrée pour s'occuper du trafic en provenance du réseau. Cependant, comme le protocole TCP Xeno n'intervient que du côté émetteur, nous nous limitons au trafic sortant.

La figure 15 présente l'emplacement du gestionnaire de file d'attente dans le kernel Linux. Seule la file d'attente pour les paquets sortants est représentée.

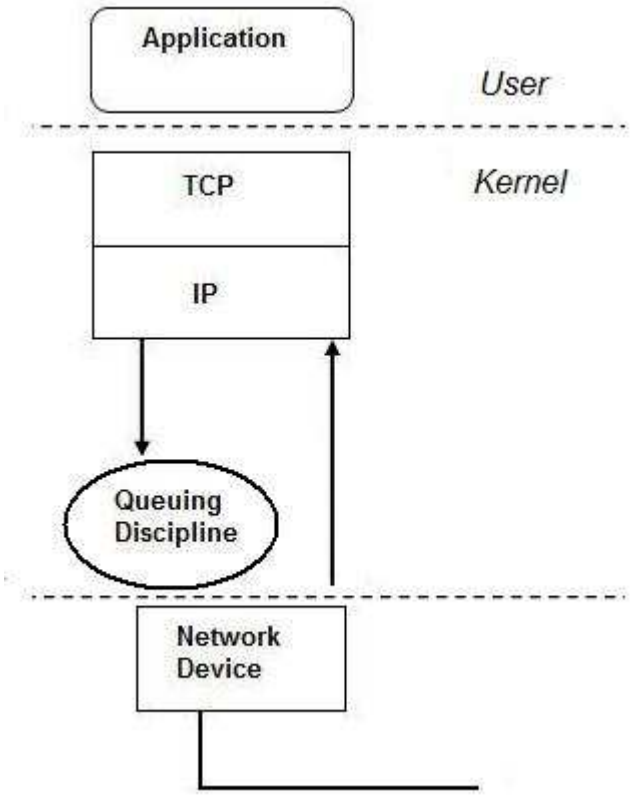

Figure 12 : Linux queuing discipline [10]

La gestion des files d'attentes permet de contrôler la façon dont les données sont envoyées. Ainsi, un gestionnaire de mise en files d'attentes (queueing discipline) détermine l'ordre de sortie des paquets et permet de retarder et/ou supprimer certains paquets. Il s'agit d'une combinaison de files d'attentes et d'un algorithme en charge de décider quel paquet envoyer, et à quel moment [6].

Un gestionnaire de mise en files d'attente peut être simple (sans classes) ou divisé en classes qui à leur tour contiennent des gestionnaires de mise en files d'attente.

Depuis la version 2.2 du kernel de Linux, un outil « TC » pour traffic control, qui permet de classifier et de filtrer des paquets a été intégré dans le kernel.

5.1 Classification et filtrage de paquets

5.1.1 Classification

Un gestionnaire de mise en file d'attente peut avoir beaucoup de classes, qui lui sont attachées. Cela permet de pouvoir appliquer un traitement différent aux paquets traversant la file d'attente. Une classe peut à son tour se voir ajouter plusieurs classes (sous classes).

Lorsqu'on crée une classe, un gestionnaire lié à cette classe est aussi créé par défaut. Et si on ajoute une/des sous classe(s) à une classe, son gestionnaire de mise en file d'attentes est automatiquement supprimé.

Une classe terminale est une classe qui ne possède pas de classes enfants. Seul un gestionnaire de mise en file d'attente est attaché à cette classe [6].

Pour construire un arbre de classe, on doit au préalable choisir d'utiliser un gestionnaire de mise en files d'attente (racine), supportant la subdivision en classes (classful).

5.1.1.1 Commandes générales:

Nous allons utiliser l'outil de commande en ligne « tc » (traffic control) qui fait partie du package d'outils **iproute2**. Les commandes tc utilisent des librairies partagées et des fichiers de données définies dans le répertoire /usr/lib/tc [8].

Génération d'un gestionnaire de file d'attente racine (root qdisc):

tc qdisc add dev DEV handle 1 : root QDISC [PARAMETER]

Génération d'un gestionnaire de file d'attente non racine:

tc qdisc add dev DEV parent PARENTID handle HANDLEID QDISC [PARAMETER]

Génération d'une classe :

tc class add dev DEV parent PARENTID classid CLASSID QDISC [PARAMETER]

Avec :

DEV: interface par où passent les paquets.

PARENTID identifiant de la classe à laquelle le gestionnaire de mise en files d'attente est attaché ex. X :Y

HANDLEID identifiant unique qui caractérise un gestionnaire de mise en files d'attente ex. X :

CLASSID identifiant unique qui caractérise une classe ex. X :Y

QDISC type du gestionnaire de mise en files d'attente attaché à la classe

PARAMETER paramètre spécifique à un gestionnaire de mise en files d'attente

Dans l'exemple suivant nous utiliserons HTB (Hierarchical Token Bucket), un des types de gestionnaire de mise en files d'attente pour la construction d'une structure arborescente de mise en file d'attente, car il permet une subdivision en sous classe (classful) et est simple à utiliser.

La figure 5 présente un arbre de file d'attente avec trois feuilles (classes) et une file de mise en file d'attente racine :

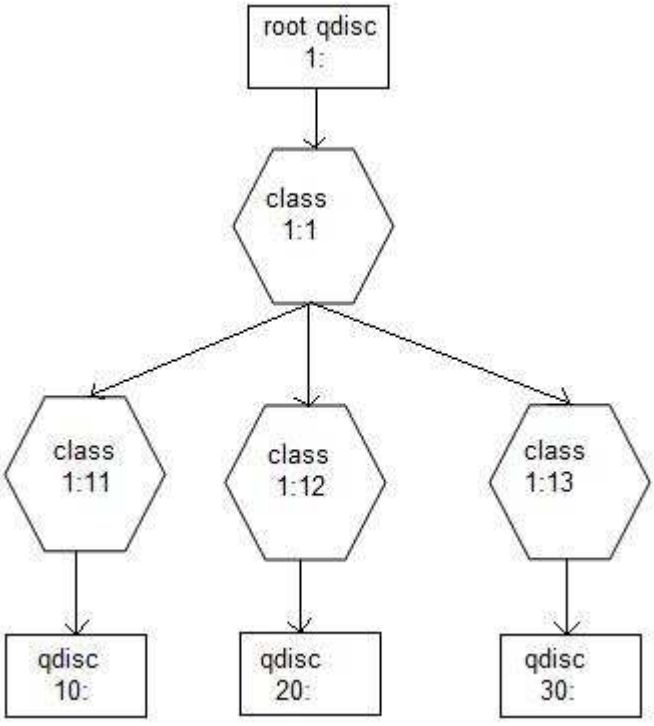

Figure 13: qdisc hierarchy [7]

D'abord une file d'attente racine est créée :

```
tc qdisc add dev eth1 handle 1 : root htb
```

Puis, une classe racine et ses trois classes filles sont créées :

```
tc class add dev eth1 parent 1 : classid 1:1 htb

tc class add dev eth1 parent 1:1 classid 1:11 htb

tc class add dev eth1 parent 1:1 classid 1:12 htb

tc class add dev eth1 parent 1:1 classid 1:13 htb
```

Le parentid d'une classe est égal au classid de son parent (classe ou file d'attente racine).

Ensuite une file d'attente (netem) peut être ajoutée à chaque classe feuille :

```
tc qdisc add dev eth1 parent 1:11 handle 10: netem

tc qdisc add dev eth1 parent 1:12 handle 20: netem

tc qdisc add dev eth1 parent 1:13 handle 30: netem
```

5.1.2 Filtrage

Une fois la file d'attente repartie en plusieurs classes et éventuellement sous classes, l'opération de filtrage consiste à associer un filtre (critères qu'un paquet doit remplir pour pouvoir passer) à une classe. Les paquets sont évalués au fil et à mesure qu'ils entrent dans la file d'attente et passent par le premier filtre dont les critères sont vérifiés.

Il existe plusieurs possibilités de faire le filtrage. Cependant, les commandes de filtre peuvent être reparties en deux groupes [7] :

- Filtrage simple et
- Filtrage complexe

Le filtrage simple permet la création de filtres qui évaluent des champs dont l'emplacement dans le paquet reste constant. Cela implique que l'entête IP du paquet ait une taille constante soit 20 octets et par conséquent, ne doit contenir aucun champ optionnel.

Par contre, le filtrage complexe est adapté aux paquets dont l'entête IP peut contenir le champ « options ». C'est-à-dire que la taille de l'entête IP est variable. Par conséquent, on ne peut pas prévoir à partir d'où commence le champ pour les données (charge utile pour le protocole IP) [9].

Ainsi, un filtre basé sur l'entête TCP ou UDP doit utiliser le filtrage complexe car l'emplacement des champs de l'entête TCP/UDP dépend de la présence ou non du champ « options » dans l'entête IP. Il en résulte que, le champ « Header length » de l'entête IP doit être analysé afin de connaitre l'emplacement exact du début de la couche supérieure (TCP/UDP).

Bien que le filtrage complexe soit capable de filtrer les paquets sur base de leur entête TCP, leur complexité d'utilisation nous encourage à utiliser plutôt le filtrage simple qui base son évaluation sur les champs d'un paquet IP, dont l'emplacement n'est pas influencé par la présence ou non de champ « options » dans son entête.

Cela explique également le choix, évoqué au chapitre précédent, de marquer le champ TOS de l'entête IP du paquet à supprimer plutôt que un de ses champs de l'entête TCP (unused par exemple).

Rappelons que les deux champs sont accessibles à partir de la structure **sk_buff,** représentant celle d'un paquet.

5.1.2.1 Structure de commande

Ajout d' un filtre :

tc filter add dev DEV protocol PROTO parent ID prio PRIO FILTER match SELECTOR [FIELD] PATTERN MASK [at OFFSET] flowid FLOWID

Suppression d'un filtre :

tc filter del DEV eth1 protocol PROTO parent ID prio PRIO

avec,

DEV: interface par où passent les paquets, ex. eth1.

PROTO: protocole sur lequel le filtre opère, ex. ip

ID: id de la classe à la quelle le filtre est attaché

PRIO : ordre de vérification des filtres. Un filtre avec le plus petit nombre a plus de priorité.

FILTER : type de filtre utilisé. On utilisera le filtre u32 dont les critères de filtrage sont basés sur les champs de paquet.

SELECTOR : dépend du type de filtre, ex pour u32 : u32, u16, u8, ip, ip6

FIELD : nom du champ à comparer (seulement pour les sélecteurs ip et ip6)

PATTERN : valeur du champ spécifié (décimale ou hexadécimale)

MASK : indique les bits qui sont comparés

OFFSET ; la comparaison commence à partir de PROTO + OFFSET octets

FLOWID : référence la classe à la quelle le filtre est attaché

5.1.3 Application du filtre pour valider le protocole TCP Xeno

Dans notre cas d'étude nous aurons besoin d'une seule classe liée à un filtre qui ne laissera pas passer des paquets marqués.

Par classification et filtrage on aurait une configuration représentée par la figure 17 ci-dessous:

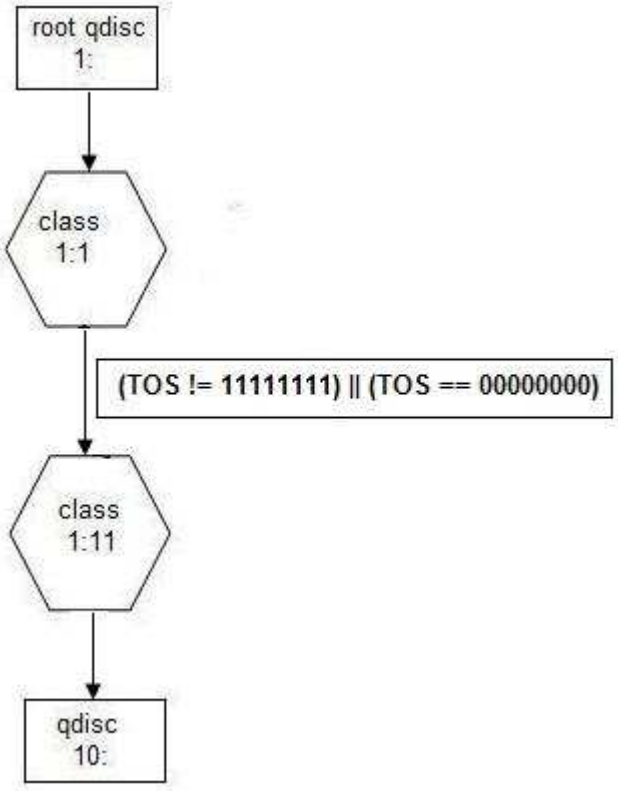

Figure 14 : Suppression de paquets dont le contenu du champ TOS vaut 11111111.

D’abord une file d’attente racine est créée :

tc qdisc add dev eth1 handle 1 : root htb

Puis, une classe racine et une classe fille est créées :

tc class add dev eth1 parent 1 : classid 1:1 htb

tc class add dev eth1 parent 1:1 classid 1:11 htb

Ensuite une file d’attente (netem) est ajoutée à la classe feuille 1:11 de façon suivante:

tc qdisc add dev eth1 parent 1:11 handle 10: netem

Un filtre doit être également ajouté à la classe 1:11 pour ne laisser passer que les paquets dont le contenu du champ TOS est différent de 11111111, soit égal à 00000000.

tc filter add dev eth1 protocol ip parent 1:11 prio 1 u32 match ip tos 15 0xff flowid 1:11

Chapitre 6

6 Intégration de l'évaluation des 5 scenarios de validation dans l'implémentation du protocole TCP Xeno

6.1 Scenarios de validation

Dans ce travail de validation du protocole TCP Xeno, il sera question d'analyser le comportement de ce protocole dans 5 scénarios de perte, où il réagit différemment par rapport à d'autres protocoles TCP, et le comparer avec ceux de ces derniers, notamment GTCP qui, comme TCP Xeno, supporte la réservation de largeur de bande.

Si on considère k le nombre de pertes survenant pendant la période de récupération de ces pertes [2], les 5 scénarios de perte possibles pour TCP Xeno sont :

1) 3dupACK, N<β,k<4/5 $cwnd_{BE}$ [1]
2) 3dupACK, N≥β,k<4/5 $cwnd_{BE}$
3) 3dupACK, k>4/5 $cwnd_{BE}$
4) Timeout, N≥β
5) Timeout, N<β

Etant donné que les expériences de validation vont se dérouler dans un labo (réseau local), le comportement d'un réseau réel sera simulé à l'aide de Netem, outil qui permet de provoquer des situations de pertes, retard, duplication, corruption et/ou re-ordonnancement de paquets.

De plus, depuis le kernel 2.2, Linux dispose d'une option de filtre avancé qui permet de filtrer des paquets sur base de divers critères. Ainsi, on peut filtrer le trafic d'un réseau selon la valeur d'un champ des paquets. C'est cette option qui sera utilisée pour les scenarios de perte se produisant à des moments précis, et qui ne peuvent pas être générés par simple simulation basique de pertes, retard, corruption …

Pour se faire :

- On doit pouvoir vérifier les conditions de réalisation de chaque scenario de perte.

- On doit pouvoir marquer un (des) paquet(s) à supprimer à un moment donné, pour réaliser un scénario. Pour cela, on va attribuer à un champ du paquet à supprimer une valeur que l'option de filtre avancé vérifiera par la suite afin de supprimer ce paquet.

Les champs ***TOS*** et ***unused*** se trouvant respectivement dans l'entête iPV4 et TCP peuvent être utilisés à cette fin.

Comme la vérification des conditions de réalisation de certains scenarios de perte exige une intervention sur un champ de paquets, c'est l'implémentation côté émetteur du module en charge du protocole de transport qui, sur base de la structure de paquets et des différents paramètres définissant l'état du réseau peut effectuer cette tâche.

Il s'en suit que les modifications doivent être apportées à l'implémentation du protocole TCP Xeno pour qu'il puisse assurer ces nouvelles fonctionnalités de marquage de champ de paquets et de vérification de survenance des conditions de réalisation des différents scenarios de perte.

Cependant, pour plus de clarté, à chaque scenario de perte ***i*** exigeant une intervention au niveau d'un champ de paquet, sera associée une implémentation de protocole « ***TCP Xeno scenario_i*** » qui en plus du fonctionnement original de TCP Xeno, permet de tester la survenance de ce scenario de perte.

On remarque que certains scenarios nécessitent l'utilisation du filtre avancé (intervention au niveau du champ *«DSCP ou unused »* des paquets à supprimer) plus que d' autres.

En effet, les scénarios 1 et 2 (3dupACK, N < β, k < 4/5 $cwnd_{BE}$ et 3dupACK, N ≥ β,k < 4/5 $cwnd_{BE}$) peuvent être réalisés en causant une perte d'un petit pourcentage de paquets k tel que k < 4/5 $cwnd_{BE}$ et en vérifiant que N < β pour le scenario 1, et N ≥ β pour le scenario 2.

Quant au scénarios 3, on doit s'assurer que k>4/5 $cwnd_{BE}$ au moment de la production de 3dupACK.

Afin pour les scénarios 4 et 5, on doit respectivement vérifier N ≥ β et N < β au moment où se produit un Timeout.

Pour plus de précision, l'implémentation du protocole TCP Xeno sera adaptée en fonction de chaque scenario à vérifier.

Selon les scénarios on aura donc les implémentations du protocole TCP Xeno suivantes:

- TCP Xeno scenario_1 pour le scenario 3dupACK, N< β, k <4/5 $cwnd_{BE}$
- TCP Xeno scenario_2 pour le scenario 3dupACK, N ≥ β, k < 4/5 $cwnd_{BE}$
- TCP Xeno scenario_3 pour le scenario 3dupACK, k > 4/5 $cwnd_{BE}$
- TCP Xeno scenario_4 pour le scenario Timeout, N ≥ β
- TCP Xeno scenario_5 pour le scenario Timeout, N < β

La figure ci- dessous représente le comportement estimé de TCP_Xeno. Il s'agit de l'association des comportements de GTCP (figure 9) et de TCP Veno (figure 8).

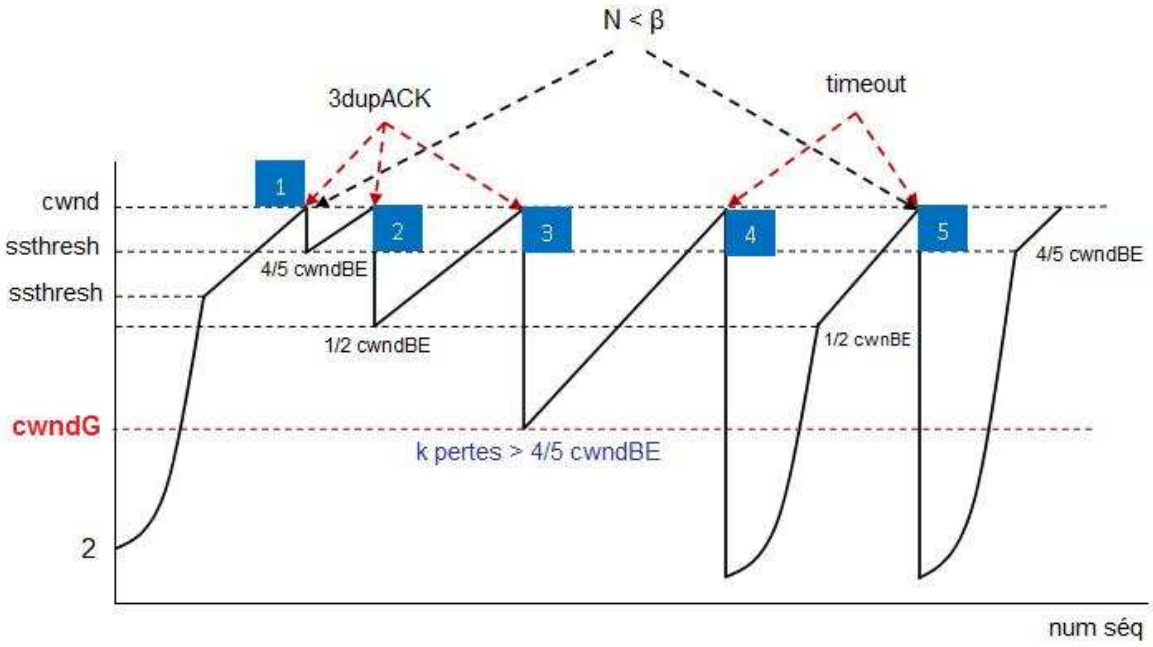

Figure 15 : Comportement du protocole GTCP [3]

Les chiffres dans la figure 18 font référence à chacun des cinq scénarios de validation.

6.2 Création de scenarios

3dupACK sont émis lorsque survient une perte de paquet(s), que certains paquets (plus exactement 3) qui le(s) suivent parviennent à atteindre le récepteur, et que les ACK associés à ces paquets reviennent à l'émetteur, avant que le RTO n'expire.

Pour ce faire RTO doit être supérieur à la somme du temps qu'il faut à trois paquets pour transiter depuis la source jusqu' à la destination et le temps qu'il faut à leurs dupACK pour quitter la destination et revenir à la source.

La figure 19 présente un scenario où on a 3dupACK. Quatre paquets sont émis. Le premier est marqué 11111111 dans son champ TOS pour être supprimé par Netem (comme on le décrira dans la partie réservée aux structures de données intervenant dans la définition des structures de paquet).

Les trois suivants passent et déclenchent au récepteur l'émission de 3 dupACK.

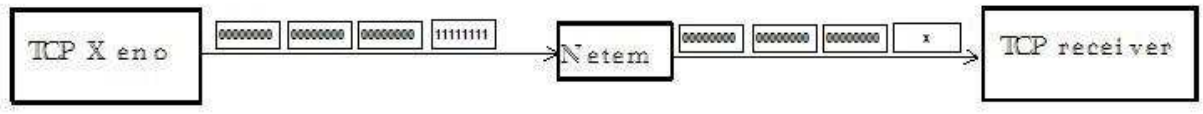

Figure 16 : scenario 3dupACK

Un Timeout se produit lorsqu'après avoir envoyé un paquet p, un temps t ≥ RTO s'écoule sans que l'émetteur ne reçoive un ACK ni pour le paquet p ni pour un éventuel paquet p+1 envoyé après.

La figure 20 présente un scenario où on a un Timeout. Bien que quatre paquets soient représentés sur la figure, en réalité le nombre de paquets successifs à supprimer doit être tel que l'émetteur attende l'ACK un temps t ≥ RTO comme décrit ci-dessus.

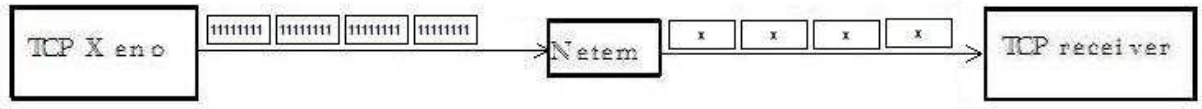

Figure 17 : scenario, timeout

Les contraintes liées à N, β, k et $cwnd_{BE}$ peuvent être vérifiées par le biais de paramètres décrivant l'état du réseau. Ces paramètres sont des membres des structures **sock** et **sk_buff** utilisées lors de l'implémentation de TCP Xeno. Les structures sock et sk_buff seront décrites dans le chapitre suivant.

6.2.1 Scenario1 (3dupACK, $N < \beta$, $k < 4/5$ $cwnd_{BE}$)

Le nombre N qui désigne les paquets accumulés dans la pile d'attente peut être calculé à partir du numéro de séquence du dernier paquet envoyé et le numéro de séquence du dernier ACK reçu.

Si LastSeq désigne le numéro de séquence du dernier paquet envoyé, et LastACK le numéro de séquence du dernier ACK reçu,

N= *LastSeq - LastACK* .

Pour définir le nombre de paquets « outstanding » sur le réseau, Linux utilise les équations [12] suivantes :

In_flight = packets_out – left_out + retrans_out

Où:

left_out = sacked_out + lost_out

avec:

packets_out : nombre de segments déjà transmis et non encore acquités

sacked_out : nombre de segments acquités

lost_out : une estimation du nombre de segments perdus dans le réseau, et

retrans_out : nombre de segments retransmis

Le problème qui se pose est surtout de contrôler N vis-à-vis de β.

Pour trouver des éléments de réponse on remarque que La valeur de N est liée au temps de transmission de paquets. On peut donc faire varier N en retardant les paquets à l'aide de l'outil netem.

Aussi, on peut faire varier N à l'aide de l'outil de génération de trafic (TC), en jouant sur la quantité de trafic émise dans le réseau.

La fenêtre de congestion qui varie selon les algorithmes Slow Start et Addition Increase – Multiplicative Decrease influence également la valeur N.

Le nombre k de paquets perdus est comparé à 4/5 $cwnd_{BE}$ (partie Best Effort de la fenêtre de congestion).

Il en résulte qu'une fois la contrainte liée à N vérifiée comme décrit ci-dessus, et que la partie Best Effort existe, pour réaliser ce scenario, il suffit de provoquer la perte d'un paquet en mettant dans son champ TOS une valeur que Netem vérifiera afin de le supprimer.

6.2.2 Scenario2 (3dupACK, N ≥ β, k < 4/5 cwndBE)

Ce scenario est semblable au précédent, à la seule différence que le protocole TCP Xeno scenario_2 qui lui est associé doit assurer que N≥β, au moment de la production de 3dupACK.

6.2.3 Scenario3 (3dupACK, k > 4/5 $cwnd_{BE}$)

Ce scenario se produit si TCP le nombre de k paquets perdus, est supérieur à 4/5 $cwnd_{BE}$ au moment où surviennent 3 dupACK.

Pour que cela se produise, la valeur du RTO doit être suffisamment grande pour éviter qu'un Timeout se produise avant 3 dupACK.

Par conséquent, plus k est grand plus il est probable qu'un Timeout se produise avant les 3 dupACK étant donné que les k paquets perdus sont consécutifs, ce qui éloignerait la réalisation du scénario.

Il en découle qu'une valeur de k proche de $cwnd_{BE}$ est la meilleure pour ce scénario.

On remarque également que le temps de transmission de 4/5 $cwnd_{BE}$ + 1 paquet et des ACK associés doit être inférieur à RTO pour que ce scenario soit possible.

6.2.4 Scenario4 (Timeout, N ≥ β)

Ce scenario doit pouvoir vérifier l'état où N ≥ β et à l'instant même provoquer un timeout. Le timeout se produit lorsqu'après avoir envoyé un paquet, un temps équivalent à RTO s'écoule sans que la source ne reçoive un ACK.

Pour générer un Timeout, on peut soit :

- Supprimer suffisamment de paquets consécutifs de telle sorte que la somme de leur temps de transmission et le temps de transmission de leurs ACK soit supérieur ou égal à RTO.
- Déconnecter le câble réseau pendant un petit moment mais supérieur à RTO.

Cependant, cette méthode est à écarter, car elle ne permet pas de vérifier les contraintes $N \geq \beta$ ou $N < \beta$. En plus cela suppose que les tests sont menés dans une infrastructure câblée alors que TCP Xeno vise le sans fil.

6.2.5 Scenario5 (Timeout, N < β)

Ce scenario est similaire au précédent. Le timeout ne doit être provoqué qu'après s'être assuré que $N < \beta$.

Durant l'analyse de conditions de réalisation de différents scenarios de perte, il s'est avéré nécessaire de marquer des paquets à supprimer. Ce marquage qui se réalise par la modification d'une valeur d'un champ d'un paquet demande au préalable la connaissance de la structure d'un paquet.

6.3 Scenarios de validation revisités

Dans ce chapitre nous avons décrit les 5 scenarios de validation à savoir :

1) 3dupACK, N<β,k<4/5 $cwnd_{BE}$ [1]
2) 3dupACK, N≥β,k<4/5 $cwnd_{BE}$
3) 3dupACK, k>4/5 $cwnd_{BE}$
4) Timeout, N≥β
5) Timeout, N<β

Nous avons également indiqué que la production d'un de ces scenarios de validation exige la modification de la valeur du champ TOS d'un ou de plusieurs paquets (représenté(s) par la structure **sk_buff**) se trouvant dans la file d'attente.

Cette file d'attente contient aussi bien des paquets non encore envoyés que des paquets déjà envoyés et qui attendent d'être acquittés.

Pour supprimer un paquet, on doit écarter le risque de marquer un paquet déjà envoyé en attente d'un acquittement, ce qui serait sans intérêt par rapport à un scenario de perte à valider.

Pour cela, on veillera à s'assurer que le numéro de séquence du paquet à marquer (champ « Seq » de la structure de l'entête TCP) soit supérieur au numéro de séquence du dernier paquet envoyé qui est fourni dans la structure **tcp_sock**. Cette exigence est valable pour tous les scenarios de validation.

La structure sock dispose d'un champ **sk_send_head** qui indique le paquet à envoyer parmi des paquets de la file d'attente non encore envoyés.

Si on ajoute comme condition la vérification que le numéro de séquence du paquet à marquer soit supérieur à celui du paquet pointé par sk_send_head, on garantit de ne pas marquer un paquet déjà envoyé, car sk_send_head ne tient pas compte de paquets à retransmettre.

Lorsqu'un Ack arrive du récepteur, le paquet acquitté est supprimé de la file d'attente, la fenêtre de réception (receive window) est élargie, et des nouveaux paquets peuvent être envoyés à partir de sk_send_head.

Signalons que lorsque tous les paquets se trouvant dans la file d'attente de sortie (`sk_write_queue`) ont déjà été envoyés au moins une fois (ils attendent leurs acquittements pour être supprimés de la file d'attente), sk_send_head vaut NULL.

La figure ci-dessous présente la relation entre la structures sock, le champ sk_send_head, la file `sk_write_queue et la strucure sk_buff :`

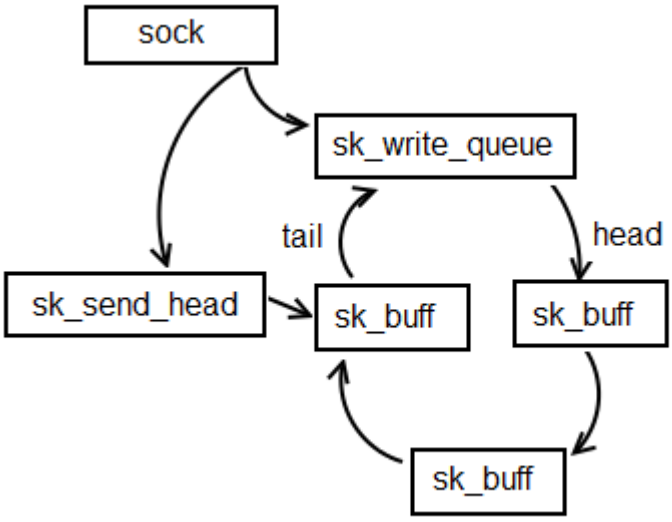

Figure 18 : Relation entre la structure sock, la file sk_write_queue, le champ sk_send_head et la structure sk_buff [11]

Si le premier et le deuxième scenario peuvent être produits simplement en supprimant un paquet une fois que les conditions liées à N sont vérifiées (N<β et N≥β, respectivement pour le premier et le deuxième scenario), les choses se compliquent un peu pour les scenarios trois, quatre et cinq.

Pour le troisième scenario (3dupACK, k>4/5 $cwnd_{BE}$), on se limite à k= $cwnd_{BE}$ comme on l'a remarqué dans la section 4.3.3, sur les scenarios de validation.

Il reste à s'assurer que pendant la perte de k paquets consécutifs, le timeout ne se produise avant la survenance de 3dupACK. On remarque également que ce scenario peut survenir aussi bien pendant la phase « slow start » que pendant la phase de « congestion avoidance ».

Quant au quatrième scenario (Timeout, N≥β) et cinquième (Timeout, N<β), une fois l'exigence liée à N vérifiée, on doit produire un Timeout.

Commençons par le troisième scenario pour finir par le quatrième et le cinquième.

Partant de la relation relative à la détermination du timeout tel que définie par Kurose dans le chapitre réservé à la couche transport [9, p 339] :

RTO= *EstimatedRTT + 4 * DevRT*, on ne peut pas toujours affirmer que pendant la suppression de k= cwnd$_{BE}$ paquets le Timeout ne se produise avant 3dupACK. Plus cwnd$_{BE}$ est grande et/ou que les cwnd$_{BE}$ paquets sont envoyés de façon dispersée (pas en rafale) plus le risque de produire un timeout plutôt que 3dupACK augmente.

D'où la nécessité de raffiner le scenario trois pour tenir compte du fait que le temps de transmission de k paquets et de réception de trois ACK puisse être inférieur au Timeout.

Remarquons que l'estimation du temps de transmission de k paquets et de réception de trois ACK dépend de la présence ou non de paquets déjà envoyés dans le réseau en attente d'accusé de réception (outstanding paquets).

Au cas où il n'y a pas de paquets en attente (toute la fenêtre de congestion est libre voir Fig 22) ou lorsque la somme de W paquets déjà envoyés durant la transmission de la fenêtre en cours et de k+3 paquets est inférieure à cwnd (W + k+3 ≤ cwnd voir Fig 23), étant donné que cwnd = cwndG + cwnd$_{BE}$ et que nous avons choisi k = cwnd$_{BE}$, c-à-dire k < cwnd, il s'en suit que tous les k= cwnd$_{BE}$ paquets vont être envoyés l'un à la suite de l'autre sans pause.

Le temps de leur transmission et de la réception de 3dupACK est estimé à :

(cwnd$_{\mathrm{BE}}$ *+3)**$\frac{L}{R}$ *+ RTT* et ce temps doit être inférieur au RTO.

Avec

L : La taille de paquet qui est fourni par le champ « *tot_len* » de l'entête IP.

R : Le taux de transmission de la liaison entre l'émetteur et le récepteur

Les deux situations sont présentées par les figures 19 et 20 ci-dessous :

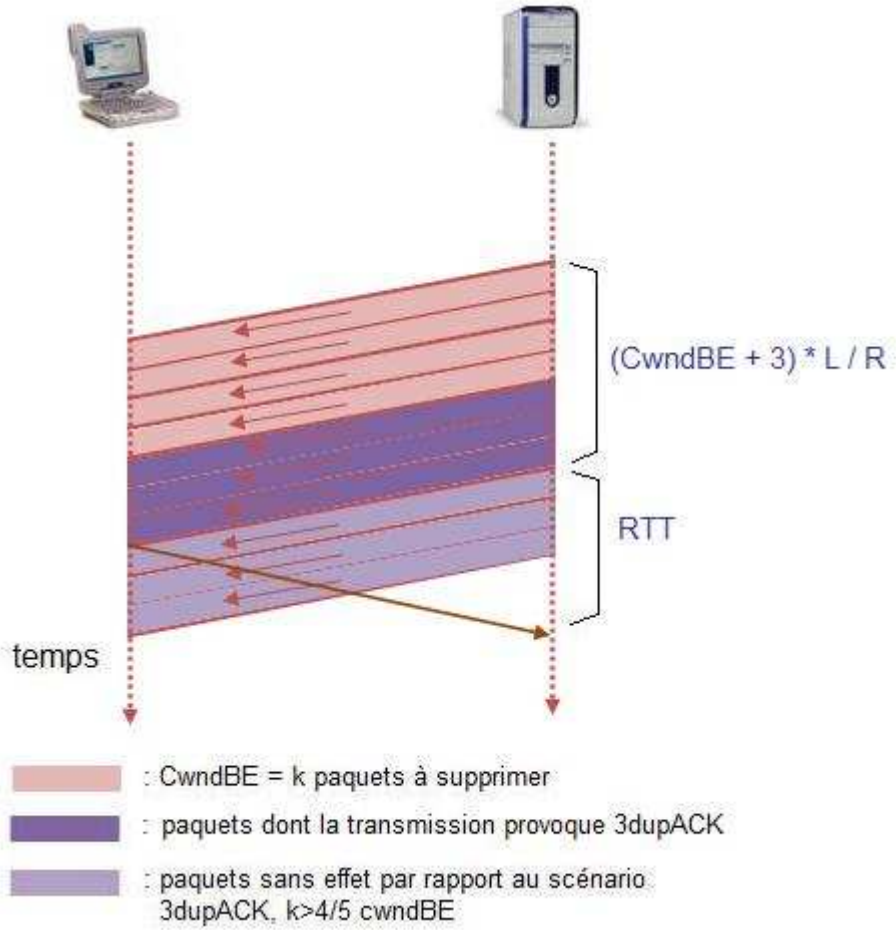

Figure 19 : Toute la fenêtre de congestion est libre, au moment où survient le scénario 3

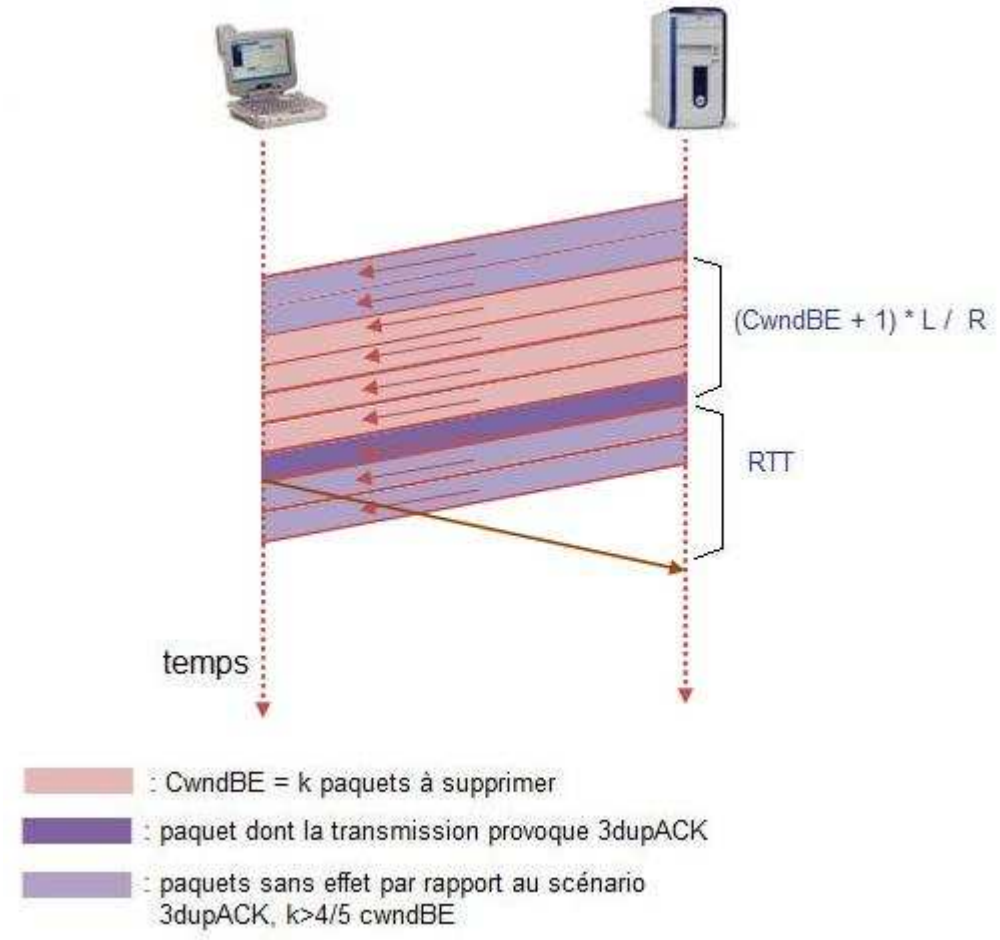

Figure 20 : W (2 sur le graphe) paquets de la fenêtre de congestion ont été émis, au moment où survient le scénario 3.

Par contre s'il existe des paquets déjà envoyés dans le réseau, et que de nouveaux paquets ne sont transmis qu'à la réception d'un ACK, la façon dont les k paquets seront transmis est liée à l'une ou l'autre des phases de l'algorithme de contrôle de congestion (Slow Start et Congestion Avoidance), on a les deux situations suivantes :

A. *Durant la phase "Slow Start" :*

Il existe une relation permettant de déterminer le temps (Latency) de transmission d'un objet de taille *O* bits sur un réseau dont la taille maximale de segment (MSS) est *L* et le taux de transmission *R*

La relation stipule que :

$$Latency = 2RTT + \frac{O}{R} + \sum_{q=1}^{Q-1}\left[\frac{L}{R} + RTT - 2^{k-1}\frac{L}{R}\right]^{+} \quad [9]$$

Avec $[x]^{+} = \max(x, O)$ et Q, le nombre de fenêtres qui constituent l'objet O.

$$Q = min\left\{q: 2^{0} + 2^{1} + \cdots + 2^{q-1} \geq \frac{O}{L}\right\}$$

A partir de cette relation nous pouvons estimer le temps de transmission de k= **cwnd**$_{BE}$ paquets plus, un k+ 1 nième paquet qui provoquera un dupACK.

On constate que :

- Le premier terme de la relation ci-dessus (2RTT) qui tient compte de l'initialisation de la connexion et la réception du premier bit par le client n'intervient pas dans notre cas, car la connexion est déjà établie durant la survenance du scénario trois, (3dupACK, k > 4/5 cwnd$_{BE}$)

- Dans le deuxième terme (O/R), l'objet O à transférer sera remplacé par

  (cwnd$_{\mathrm{BE}}$ *+1)*L* paquets à transmettre avant la production du premier dupACK.

- Afin le troisième terme $\sum_{q=1}^{Q-1}\left[\frac{L}{R}+\ RTT-\ 2^{k-1}\frac{L}{R}\right]^{+}$

  tienne compte d'éventuels temps morts que l'émetteur (serveur) peut passer sans rien émettre. Ce temps gaspillé apparaît entre une fenêtre i et une fenêtre i+1, lorsque le temps de transmission du premier paquet de la fenêtre i, et l'arrivée de son accusé de réception, c'est-à-dire $\frac{L}{R}+\ RTT$, est supérieur au temps de transmission de tous les paquets de la fenêtre i. C'est-à-dire que le serveur finit de transmettre tous les paquets de la fenêtre i avant de recevoir l'accusé de réception du premier paquet de cette même fenêtre.

  Dans le cas contraire, la transmission de la fenêtre i+1 a immédiatement suivi celle de la fenêtre i sans arrêt, et il n'y a pas de temps d'attente mort. Ce cas est similaire à celui représenté par la figure 19 ci-dessus.

  Par contre, s'agissant du scenario trois (3 dupACK, k > 4/5 cwnd$_{BE}$), le nombre de paquets *cwnd*$_{\mathrm{BE}}$ *+1* étant inférieur à la fenêtre de congestion (cwnd), si jamais le problème de temps mort entre des fenêtres se pose, il ne peut apparaître qu'entre deux fenêtres successives i et i+1.

  Cela correspondrait au cas où une partie de *cwnd*$_{\mathrm{BE}}$ *+1* paquets est envoyée pendant la transmission de la fenêtre i alors qu'une autre partie est envoyée pendant la transmission d'i+1.

  On en déduit que le temps d'attente vaudrait $\frac{L}{R}+\ RTT-\ Cwndi*\frac{L}{R}$

  avec Cwndi, le nombre de paquets constituant la fenêtre i.

La figure ci-dessous présente une situation où le serveur connaît un temps mort pendant la transmission de k = cwnd paquets :

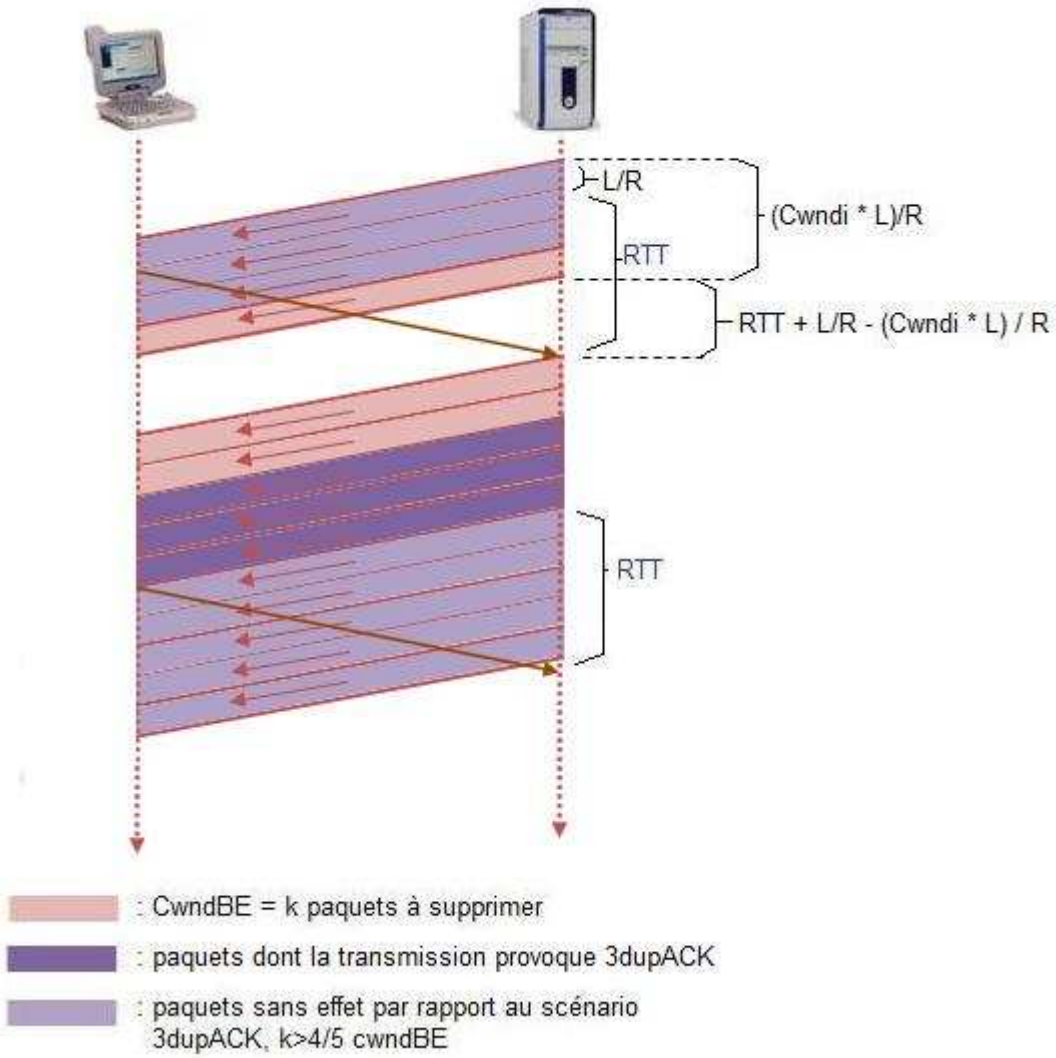

Figure 21 : La transmission de k + 1 paquets avec k = cwndBE est repartie sur deux fenêtres (Cwndi et Cwndi+1), et $\frac{L}{R} + RTT - Cwndi * \frac{L}{R} > 0$

Ayant analysé les trois termes associés au calcul du temps de transmission d'un objet, on en déduit que seul le deuxième terme intervient pour estimer le temps nécessaire pour produire le scenario 3dupACK, k>4/5 $cwnd_{BE}$.

Ce temps est constitué du temps de transmission de $cwnd_{BE}$ +1 paquets, soit *(cwnd*$_{BE}$ *+1)** $\frac{L}{R}$, auquel on ajoute :

- éventuellement le temps d'attente entre deux fenêtres
  $\left[\frac{L}{R} + RTT - Cwndi * \frac{L}{R}\right]^{+}$

  et Le temps de transmission du premier dupack : RTT

Par conséquent, on doit s'assurer que :

(cwnd$_{BE}$ *+1)** $\frac{L}{R}$ + $\left[\frac{L}{R} + RTT - Cwndi * \frac{L}{R}\right]^{+}$ + RTT est inférieur au Timeout pour produire 3dupack avant la survenance de ce dernier.

B . Durant la phase Congestion Avoidance

A la réception d'un Ack, un nouveau paquet est envoyé et la fenêtre de congestion est augmentée de 1/cwnd. Si on fait abstraction de cette légère augmentation de la fenêtre de congestion, sachant que les différents Ack sont espacés d'une période de L/R (les paquets sont envoyés sans arrêt pendant cette phase), le temps pour transmettre *cwnd*$_{BE}$*+1* paquets et recevoir le premier dupack serait égal à :

(cwnd$_{BE}$ *+1)** $\frac{L}{R} + RTT$ et ce temps doit être inférieur à **RTO** pour garantir 3dupack.

Cette situation est similaire à celle représentée par la figure 19.

Remarquons qu'on ne tient pas compte d'éventuels temps de traitement de paquets et de la mise en file d'attente (on n'utilise pas de routeur).

Concernant les scenarios 4 (Timeout, N ≥ β) et 5 (Timeout, N < β), il faut trouver le nombre de paquets successifs qui doivent être marqués et supprimés pour produire un timeout.

Pour cela, on doit répondre à la question : pendant un temps égal à RTO, combien de paquets sont-ils envoyés ?

Si on considère que l'algorithme de contrôle de congestion est en phase de Congestion Avoidance (plus adaptée au scenario 4) c-à-dire qu'il attend la réception d'un ACK pour envoyer un nouveau paquet comme on l'a vu ci-dessus, on peut en déduire que le nombre de paquets consécutifs à supprimer pour provoquer un timeout doit être supérieur ou égal à :

$$RTO / (\frac{L}{R}) \equiv RTO * \frac{R}{L}$$

Rappelons que les ACK arrivent après chaque période de L/R.

Par contre, si l'algorithme est en phase *Slow Start* (plus adaptée au scenario 5), c'est-à-dire qu'à la réception d'un ACK l'émetteur envoie deux paquets, l'émetteur peut connaître durant cette phase une période où il n'émet pas en attendant l'arrivée de l'accusé de réception pour le premier paquet de la fenêtre précédente.

Cependant, étant donné que c'est quand l'émetteur continue de transmettre des paquets sans arrêt qu'il émet plus de paquets par unité de temps, on en déduit que si nous considérons l'émetteur fonctionnant comme tel (sans temps d'arrêt), cela couvre les cas où il connaîtrait des périodes de repos pendant la transmission.

Car, si le Timeout se produit en supprimant un certain nombre de paquets, il se produira également en supprimant un peu plus de paquets.

Ainsi, on considère que le nombre de paquets à supprimer pour provoquer un timeout doit être supérieur ou égal au nombre de paquets transmissible si l'émetteur reste occupé pendant une période égale au Timeout. Ce qui fait :

$$RTO / (\frac{L}{R}) \equiv RTO * \frac{R}{L}$$ paquets.

La valeur du RTO peut être calculée à partir des champs rttvar (smoothed mdev_max) et mdev (medium deviation) de la structure tcp_sock, selon la formule:

*RTO= rttvar + 4 * mdev* [9].

6.2 Evaluation

6.2.1 Evolution de la fenêtre de congestion dans TCP Xeno

Pour évaluer l'évolution de la fenêtre de congestion et avoir d'autres informations comme le numéro de séquence du prochain segment à envoyer, une petite modification a été apportée au code du protocole TCP Xeno, en insérant la fonction « printk() » et sa librairie « linux/kernel.h ». La fonction printk() permet d'écrire des messages dans le fichier « var/log/messages ».

En effet les fonctions de la librairie standard comme printf() ne sont pas autorisées au niveau du kernel.

Nous avons ajouté printk() dans la fonction « *tcp_xeno_cong_avoid* ()» pour afficher respectivement le numéro de séquence du prochain segment à envoyer(snd_nxt), la taille de la composante garantie de la fenêtre de congestion (cwnd_G) et la taille de la fenêtre de congestion (snd_cwnd), comme suit:

«

```
static void tcp_xeno_cong_avoid(struct sock *sk, u32 ack, u32 seq_rtt, u32 in_flight, int flag)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct xeno *xeno = inet_csk_ca(sk);

printk(KERN_INFO "[CONG_AV1]:snd_nxt:%u cwnd_G:%d snd_cwnd:%d\n", tp->snd_nxt, xeno->cwnd_G, tp->snd_cwnd);//
        …….
```

»

Le choix la fonction *tcp_xeno_cong_avoid()* est lié au fait qu'elle est appelée à chaque fois qu'un ACK arrive du récepteur ce qui nous garantit une régularité de messages.

Un fichier de 9,1MO a été envoyé au serveur « student.info.fundp.ac.be » avec le MTU respectivement fixé à 1500, 750 et 500 octets. Le taux « grate » (paramètre pour la composante garantie était fixé à 100). Tous les paquets passaient (aucune perte)

Des messages inscrits dans le fichier /var/log/messages ont été analysés à l'aide du programme Excel et les graphiques suivants ont été produits .

La figure ci-dessous montre l'évolution de la fenêtre de congestion pour MTU égale à 1500

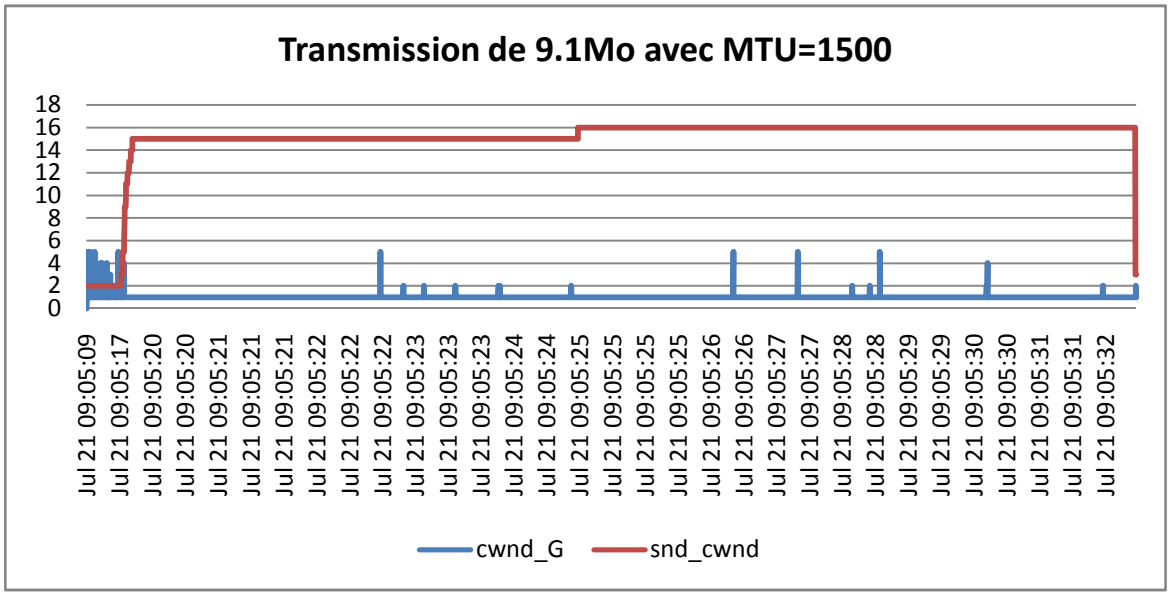

Figure 22 : Transmission par TCP Xeno de 9,1MO avec MTU = 1500

La figure ci-dessous montre l'évolution de la fenêtre de congestion pour MTU égale à 750

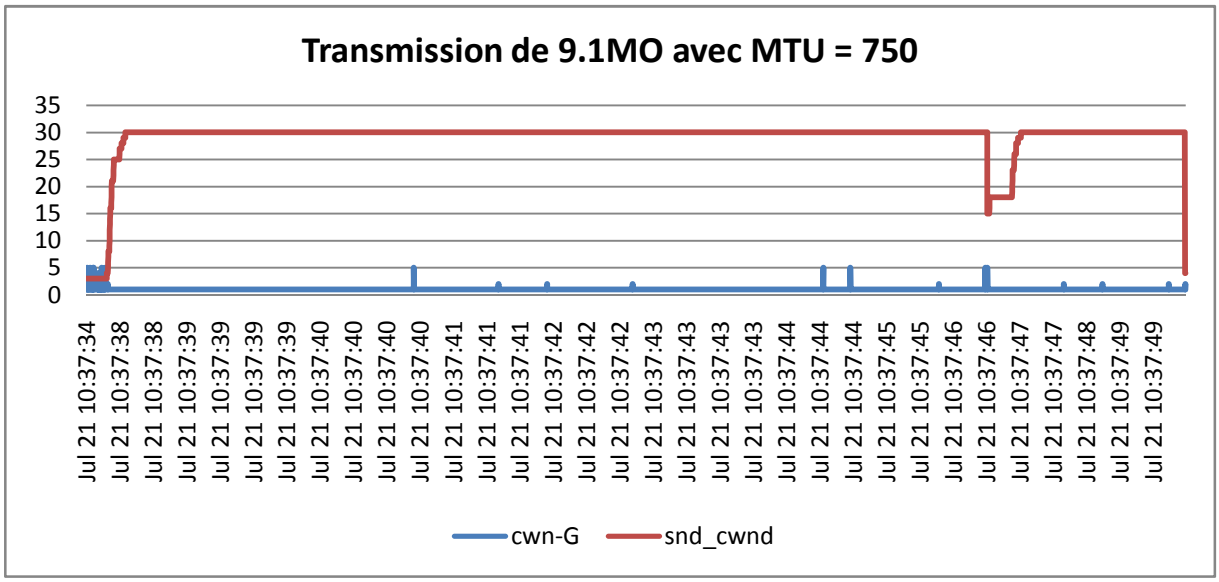

Figure 23 : Transmission par TCP Xeno de 9,1MO avec MTU = 750

La figure ci-dessous montre l'évolution de la fenêtre de congestion pour MTU égale à 500

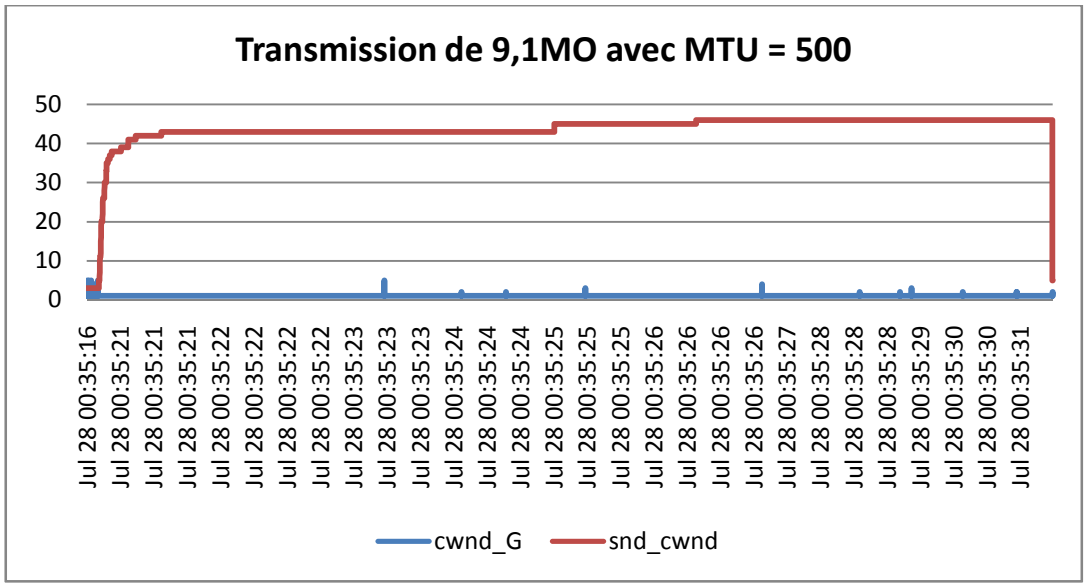

Figure 24 : Transmission par TCP Xeno de 9,1MO avec MTU = 750

On remarque que plus le MTU augmente plus la taille de la fenêtre de congestion diminue et plus le temps de transfert augmente. TCP Xeno est plus efficace lorsqu'il envoie des segments de petite taille.

Il faut aussi noter que sa partie garantie reste faible (cwnd_G = 1 la plupart du temps).

Les tailles maximales de la fenêtre de congestion semblent faibles :

16 segments à MTU = 1500, 30 à MTU = 750 et 46 à MTU = 500. Nous avons vérifié l'exactitude de ces mesures par le produit bande passante - délai. En régime, on a approximativement que

R*RTT ~= cwnd_MAX*MTU

En utilisant SpeedTest.net Mini (http://www.info.fundp.ac.be/~rco/test/), nous avons mesuré R = 60 Mbps. Avec cwnd_MAX = 16 pour un MTU = 1500 octets, on dérive RTT = 3,2 ms, ce qui est réaliste, puisque client et serveur sont sur le réseau des FUNDP.

Par ailleurs, si le RTT était aussi faible, alors cwnd_G = grate * BaseRTT = 100 * 0,003 < 1

6.2.2 Evaluation de performance

Dans l'évaluation de performance nous allons comparer l'efficacité de transmission du protocole TCP Xeno par rapport à d'autres protocoles TCP. Les protocoles choisis pour effectuer cette comparaison sont :

- GTCP qui est le concurrent direct de TCP Xeno car, comme TCP Xeno, il propose la possibilité de réservation de largeur de bande.

- TCP Veno qui bien que ne permet pas la réservation de largeur de bande, est comme TCP Xeno, adapté à fonctionner dans des réseaux sans fils

- Enfin nous allons comparer TCP Xeno avec TCP Reno, qui est actuellement le protocole TCP le plus utilisé.

6.2.2.1 Environnement et déroulement des tests d'évaluation de performance

Pendant les tests de performance un fichier de 37,1Mo a été transféré (par programme libre gFTP) au serveur « student.info.fundp.ac.be » en utilisant les quatre protocoles TCP à tour de rôle.

Signalons qu'une fois les modules de différents protocoles compilés et installés dans le kernel, on peut utiliser la commande `modprobe` pour charger tel ou tel autre protocole. Par exemple pour charger TCP Veno on ferait `modprobe tcp veno.` Pour des modules de protocoles qui peuvent accepter un paramètre comme TCP Xeno et GTCP, on fait `modprobe tcp veno grate = Nbr.`

Avec `grate` un coefficient intervenant dans le calcul de la composante garantie de la fenêtre de congestion et `Nbr` sa valeur. Rappelons que CwndBE = `grate * BaseRTT.`

TCP Reno est le protocole utilisé par défaut (si aucun autre n'est chargé).

Au cours du transfert nous avons effectué une capture de paquets avec la commande Linux « tcpdump » et enregistré le résultat de chaque capture dans un fichier.

Le fichier enregistré peut par après être analysé à l'aide du programme « Ethereal », qui détermine entre autres la durée de transfert du contenu du fichier.

Nous avons également eu recours à l'outil Nettem pour simuler des pertes durant la transmission. Le taux de la composante garantie pour TCP Xeno et GTCP a été fixé à 100.

6.2.2.2 Résultat des tests

Test1 : temps de transmission de 37,1Mo avec perte de 3%	*Test2 : temps de transmission de 37,1Mo avec perte de 10%*
TCP Reno : 2 min 2 s TCP Veno : 1 min 56 s TCP GTCP : 2 min 4 s TCP Xeno : 1 min 51 s	TCP Reno : 7 min 31 s TCP Veno : 7 min 25 s TCP GTCP : 7 min 29 s TCP Xeno : 7 min 18 s

Le tableau précédent montre que TCP Xeno est plus performant que GTCP et TCP Veno, qui comme TCP Xeno, sont plus adaptés à fonctionner dans des réseaux sans fil.

6.2.3 Evaluation de validité

Cette partie vise à prouver que TCP Xeno se comporte comme indiqué par les scénarios de la figure 10. Une analyse de chaque scénario a été effectuée au cours du chapitre 6.

Les résultats de l'analyse de performance montrent que TCP Xeno se comporte selon ses spécifications théoriques.

En effet, TCP Xeno étant l'association de GTCP et de TCP Veno, s'il est plus performant que GTCP, cela ne peut qu'être dû à une gestion intelligente des cas de pertes selon TCP Veno.

Dans l'annexe nous présentons une ébauche de fonctions qui une fois intégrées dans l'implémentation du protocole TCP Xeno, et combinées avec le mécanisme de filtrage du chapitre 5, permettraient une validation plus explicite de chaque scénario.

Chapitre 7

7 Conclusion

Au cours de ce travail de validation du protocole TCP Xeno, les tests de performance ont montré l'efficacité du protocole TCP Xeno par rapport à d'autres protocoles de contrôle de cogestion, surtout par rapport à GTCP qui comme TCP Xeno, propose la réservation de bande passante aux applications plus exigeantes.

L'implémentation de TCP Xeno montre comment à partir des fonctionnalités de différents protocoles, on peut produire un nouveau protocole de meilleure qualité.

Une analyse de 5 scénarios de perte du protocole TCP Xeno a été effectuée pour dégager les conditions de production de chaque scenario.

Le chapitre 4 a permis une compréhension de la partie du kernel Linux associé aux modules et structures intervenant dans la gestion des réseaux et à la représentation des paquets dans le gestionnaire de file d'attente du kernel, où ces paquets peuvent subir des opérations de filtrage.

A l'état actuel, l'intégration des fonctions de validation de différents scénarios dans l'implémentation du protocole TCP Xeno n'est pas encore effective, dû probablement au blocage lié au fait que l'implémentation du protocole TCP Xeno suit une structure de codage définie, où les fonctions sont préalablement définies sous forme d'interface.

Bibliographie

[1] Cheng. P. Fu, Soung C. Liew *"TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks", IEEE Journal on Selected Areas in Communications,* Feb 2003. http://www3.ntu.edu.sg/home/ascpfu/veno.pdf

[2] Yujie Zhu, Aravind Velayutham, Oyebamiji Oladeji , Raghupathy Sivakumar" Enhancing TCP for networks with guaranteed bandwidth services*", School of Electrical and Computer Engineering, Georgia Institute of Technology, 330605 Georgia Tech Station, Atlanta, GA 30332, United States, November 2006.* http://www.sciencedirect.com

[3] Christos Styliaras *"TCP Xeno: Interaction de GTCP et de TCP Veno", FUNDP, Faculté d'Informatique, Namur 2006-2007.*

[4] M. Rio et al. *"A Map of the Networking Codein Linux Kernel 2.4.20",Technical Report DataTAG-2004-1FP5/IST DataTAG Project,* Mars 2004. http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf

[5] M. Tim Jones *"Anatomy of the Linux networking tack,* From sockets to device *drivers*", *IBM,* Juin 2007. http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/

[6] http://www.linux-france.org/prj/inetdoc/guides/Advanced-routing-Howto/lartc.qdisc.html, 17/03/2009

[7] Ariane Keller, "*tc Packet Filtering and netem*", *ETH Zurich* 20 Juillet 2006 http://tcn.hypert.net/tcmanual.pdf

[8] The Linux Foundation, http://www.linuxfoundation.org/en/Net:Netem, 18 Mars 2009

[9] James F. Kurose, Keith W. Ross, *Computer networking, a top-down approach featuring the internet, International third Edition*, 2005

[10] Stephen Hemminger, "*Network Emulation with NetEm*", *Open Source Development Lab*, April 2005 http://devresources.linux-foundation.org/shemminger/netem/LCA2005_paper.pdf

[11] http://vger.kernel.org/~davem/tcp_output.html , 17/04/2007

[12] Le code source de Linux : http://www.kernel.org, 23/06/2009

[13] Omar Ait-Hellal, Eitan Altman, *Evolution of TCP : problems and enhancements,* INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

[14] http://www.opalsoft.net/qos/TCP-40.htm , 11/08/2009

[15] http://www.faqs.org/rfcs/rfc2582.html, 11/08/2009

[16] http://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf 12/08/2008

[17] René Pfeiffer – "TCP and Linux' Pluggable Congestion Control Algorithms" , http://linuxgazette.net/135/pfeiffer.html

[18] http://www.freesoft.org/CIE/Course/Section4/8.h 23/06/200909

[19] http://lhermie.homelinux.com/docs/COURS 23/06/200909

[20] http://xml.resource.org/public/rfc/html/rfc2001.html 09/08/2009

[21] http://www.eventhelix.com/RealtimeMantra/Networking/TCP Fast_Retransmit_and_Recovery.pdf 09/08/2009

[22] Dong Lin, H.T. Kung , *TCP Fast Recovery Strategies: Analysis and Improvements*, Division of Engineering and Applied Sciences Harvard University Cambridge, MA 02138 USA

[23] http://www.faqs.org/rfcs/rfc2001.html 05/08/2009

[24] Kevin Fall, Sally Floyd , *Simulation-based Comparisons of Tahoe, Reno, and SACK TCP,* Lawrence Berkeley National Laboratory One Cyclotron Road, Berkeley, CA 94720

[25] http://www-didc.lbl.gov/TCP-tuning/linux-2.6.13-tcp.txt 27/07/2009

[26] http://lwn.net/Articles/128681/ 27/07/2009